HYVISUAL: A HYBRID SYSTEM VISUAL MODELER

Authors[1]:   *Christopher Hylands*
               Edward A. Lee
               Jie Liu
               Xiaojun Liu
               Steve Neuendorffer
               Haiyang Zheng
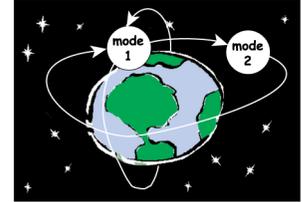
Version 2.2-beta[2]
UCB ERL Memorandum M03/1
January 28, 2003

---
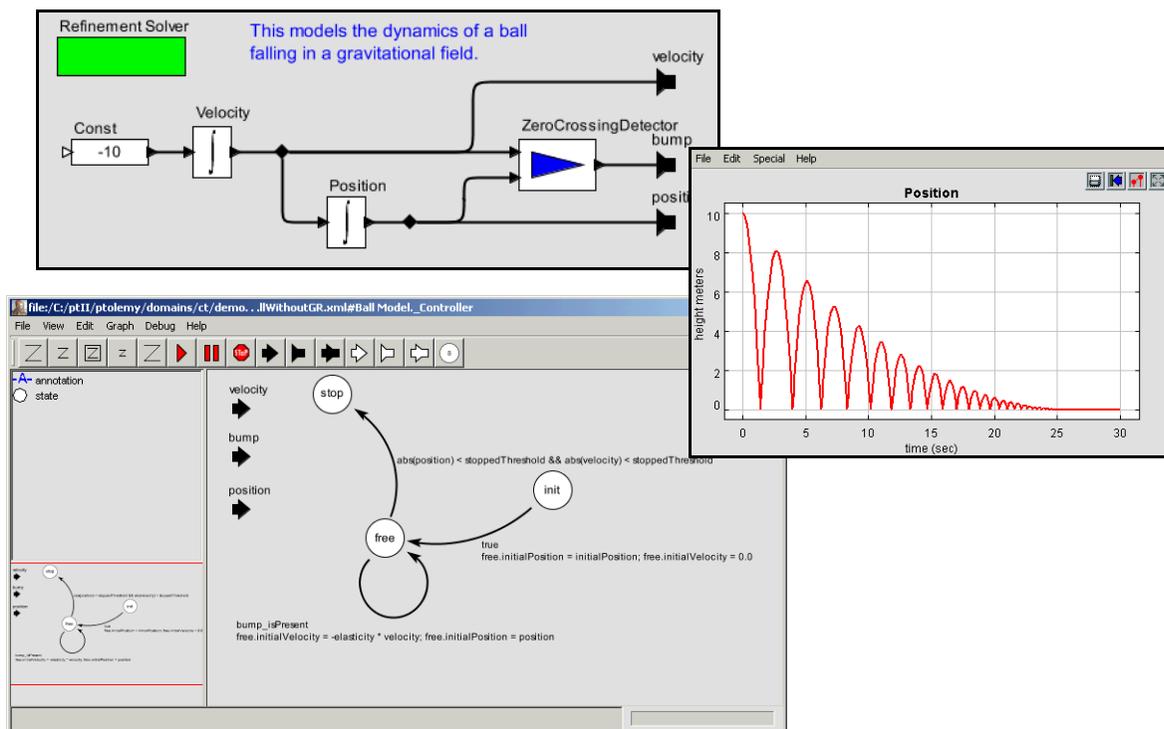
1. With contributions from the entire Ptolemy II team, but most especially John Reekie and Yuhong Xiong.
2. This is the first version of HyVisual, but the version number is selected to match the version of Ptolemy II on which it is based.

Contents

1. Introduction

The Hybrid System Visual Modeler (HyVisual) is a block-diagram editor and simulator for continuous-time dynamical systems and hybrid systems. Hybrid systems mix continuous-time dynamics, discrete events, and discrete mode changes. This visual modeler supports construction of hierarchical hybrid systems. It uses a block-diagram representation of ordinary differential equations (ODEs) to define continuous dynamics, and allows mixing of continuous-time signals with events that are discrete in time. It uses a bubble-and-arc diagram representation of finite state machines to define discrete behavior driven by mode transitions.

In this document, we describe how to graphically construct models and how to interpret the resulting models. HyVisual provides a sophisticated numerical solver that simulates the continuous-time dynamics, and effective use of the system requires at least a rudimentary understanding of the properties of the solver. This document provides a tutorial that will enable the reader to construct elaborate models and to have confidence in the results of a simulation of those models. We begin by explaining how to describe continuous-time models of classical dynamical systems, and then progress to the construction of mixed signal and hybrid systems.

The intended audience for this document is an engineer with at least a rudimentary understanding of the theory of continuous-time dynamical systems (ordinary differential equations and Laplace transform representations), who wishes to build models of such systems, and who wishes to learn about hybrid systems and build models of hybrid systems.

HyVisual is built on top of Ptolemy II, a framework supporting the construction of such domain-specific tools. See http://ptolemy.eecs.berkeley.edu for information about Ptolemy II.

1.1 Installation and Quick Start

HyVisual can be quickly downloaded and run using Web Start from the web site:

```
http://ptolemy.eecs.berkeley.edu/hyvisual
```

Once you have done this once, then you can select *HyVisual* from the Ptolemy II entry in the Start menu (if you are using a Windows system). You should then see an initial welcome window that looks something like the one in figure 1. Feel free to explore the links in this window.

To create a new model, invoke the New command in the File menu. But before doing this, it is worth understanding how a model works.

HyVisual is also available as a standalone installer for Windows (.exe file), experimental installers for other platforms, and as a CD. This document is included with the software in PDF format, so if you would like to read the HyVisual Documentation online, then you may need to install the Adobe Acrobat Reader.

HyVisual-2.2beta is available in executable format only; the source files for a later version are included with Ptolemy II 2.2, available from mid-2003. HyVisual requires Java 1.4 or later. Java 1.4.1_01 is preferred.

1.1.1 Web Start

Web Start is a tool from Sun Microsystems that makes software installation and updates particularly simple. The Web Start installation works best with Windows, but has also been tried under

Solaris, Linux and Mac OS X. The Web Start installation behaves almost exactly like a standalone installation; you can save models locally, and you need not be connected to the net after the initial installation. The Web Start tool includes a Java Runtime Environment (JRE), and the HyVisual Web Start installer checks that the proper version of the JRE is present.

1.1.2 Standard Installers

The Windows installer and the experimental installers for other platforms are shipped as a single executable. One of the Windows installers includes a 1.4.1_01 Java Runtime Environment (JRE). Note that this JRE will be installed as a private copy and will not be directly accessible by other programs. Under Windows, the installer will create a Ptolemy, HyVisual 2.2-beta menu choice in the Start menu.

1.1.3 CD

The HyVisual CD includes installers for HyVisual, the Java Runtime Environment version 1.4.1_01 and Adobe Acrobat Reader 5.1.
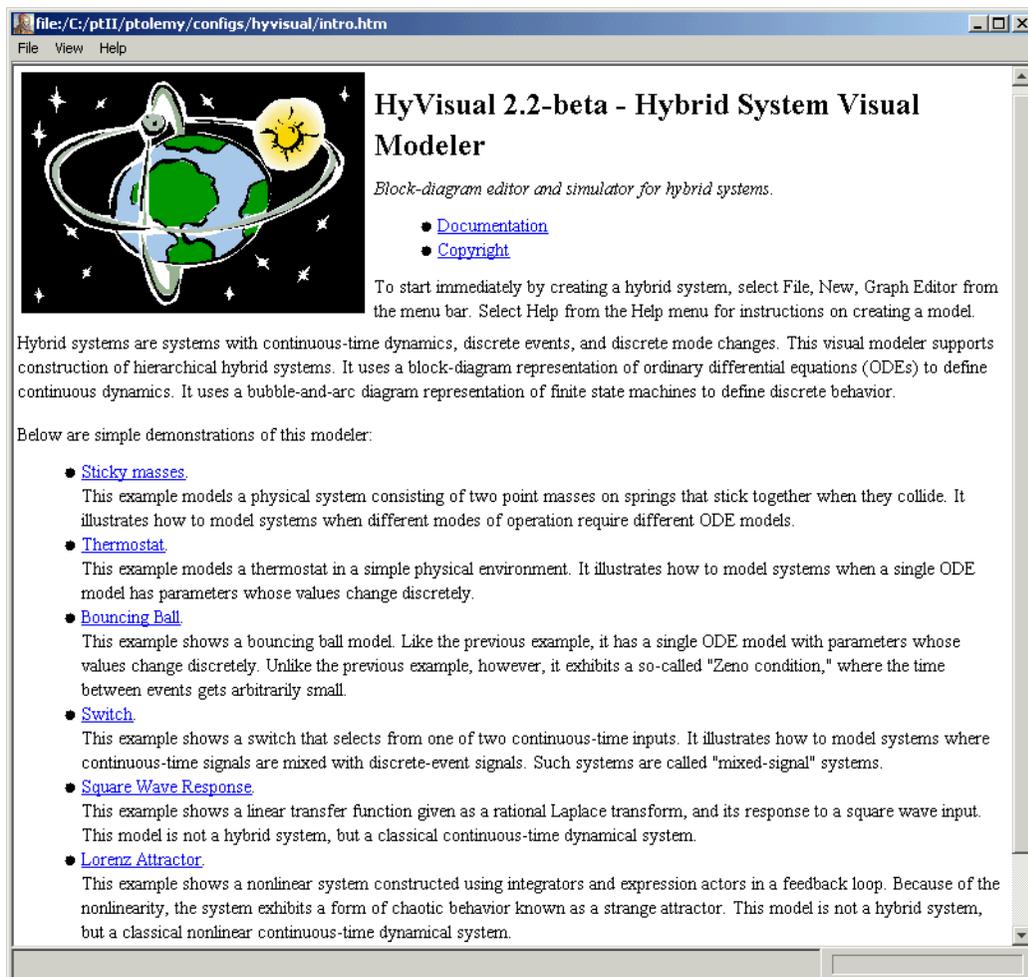
FIGURE 1. Initial welcome window.

2. Continuous-Time Dynamical Systems

In this section, we explain how to read, construct and execute models of continuous-time systems. We begin by examining a demonstration system that is accessible from the welcome window in figure 1, the Lorenz attractor.

2.1 Executing a Pre-Built Model

The Lorenz attractor model can be accessed by clicking on the link in the welcome window, which results in the window shown in figure 2. It is a block diagram representation of a set of nonlinear ordinary differential equations. The blocks with integration signs in their icons are integrators. At any given time t, their output is given by

$$x(t) = x(t_0) + \int_{t_0}^{t} \dot{x}(\tau)d\tau, \tag{1}$$

where $x(t_0)$ is the initial state of the integrator, t_0 is the start time of the model, and $\dot{x}$ is the input signal. Note that since the output is the integral of the input, then at any given time, the input is the derivative of the output,

$$\dot{x}(t) = \frac{d}{dt}x(t). \tag{2}$$

Thus, the system describes either an integral equation or a differential equation, depending on which of
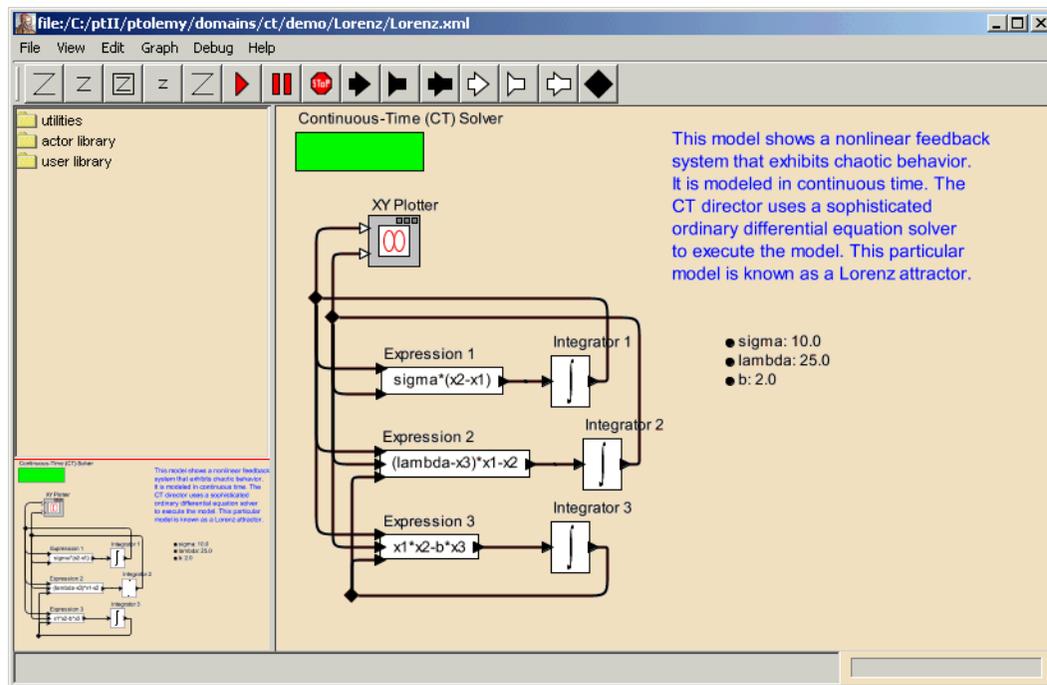

FIGURE 2. A block diagram representation of a set of nonlinear ordinary differential equations.

these two forms you use.

Let the output of the top integrator in figure 2 be x_1, the output of the middle integrator be x_2, and the output of the bottom integrator be x_3. Then the equations described by figure 2 are

$$\dot{x}_1(t) = \sigma(x_2(t) - x_1(t))$$
$$\dot{x}_2(t) = (\lambda - x_3(t))x_1(t) - x_2(t). \quad (3)$$
$$\dot{x}_3(t) = x_1(t)x_2(t) - bx_3(t)$$

For each equation, the expression on the right is implemented by an Expression actor, whose icon shows the expression. Each expression refers to parameters (such as *lambda* for λ and *sigma* for σ) and input ports of the actor (such as *x1* for x_1 and *x2* for x_2). The names of the input ports are not shown in the diagram, but if you linger over them with the mouse cursor, the name will pop up in a tooltip. The expression in each Expression actor can be edited by double clicking on the actor, and the parameter values can be edited by double clicking on the parameters, which are shown next to bullets on the right.

The integrators each also have initial values, which you can examine and change by double clicking on the corresponding integrator icon. These define the initial values of x_1, x_2, and x_3, respectively. For this example, all three are set to 1.0.

The Continuous-Time (CT) Solver, shown at the upper right, manages a simulation of the model. It contains a sophisticated ODE solver, and to use it effectively, you will need to understand some of its parameters. The parameters are accessed by double clicking on solver box, which results in the dialog shown in figure 3. The simplest of these parameters are the *startTime* and the *stopTime*, which are self-explanatory. They define the region of the time line over which a simulation will execute.

To execute the model, you can click on the run button in the toolbar (with a red triangle icon), or you can open the Run Window in the View menu. In the former case, the model executes, and the results are plotted in their own window, as shown in figure 4. What is plotted is $x_1(t)$ vs. $x_2(t)$ for values of t in between *startTime* and *stopTime*. The Run Window obtained via the View menu is shown in figure 5.
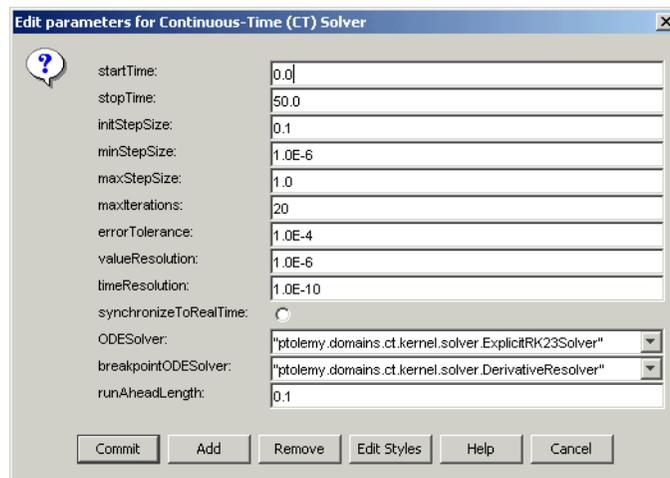
FIGURE 3. Dialog box showing solver parameters for the model in figure 2.

Like the Lorenz model, a typical continuous-time model contains integrators in feedback loops, or more elaborate blocks that realize linear and non-linear dynamical systems given abstract mathematical representations of them (such as Laplace transforms). In the next section, we will explore how to build a model from scratch.

2.2 Creating a New Model

Create a new model by selecting File, New, and Graph Editor in the welcome window. You should see something like the window shown in figure 6. On the upper left is a library of objects that can be
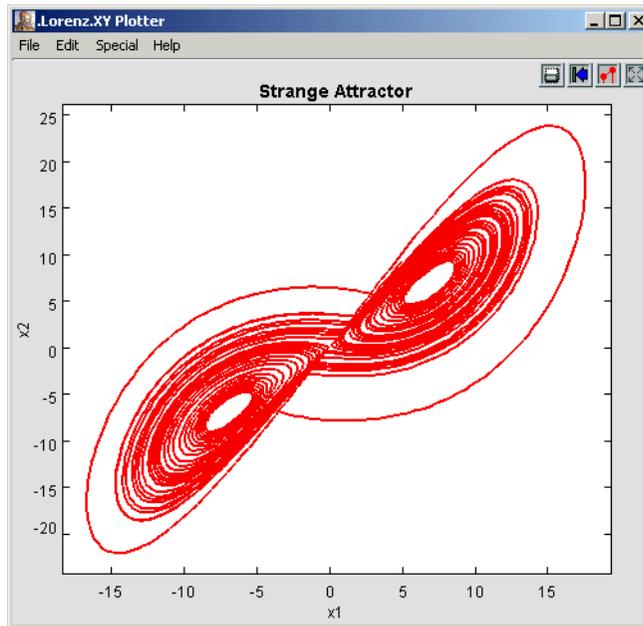
FIGURE 4. Result of running the Lorenz model using the run button in the toolbar.
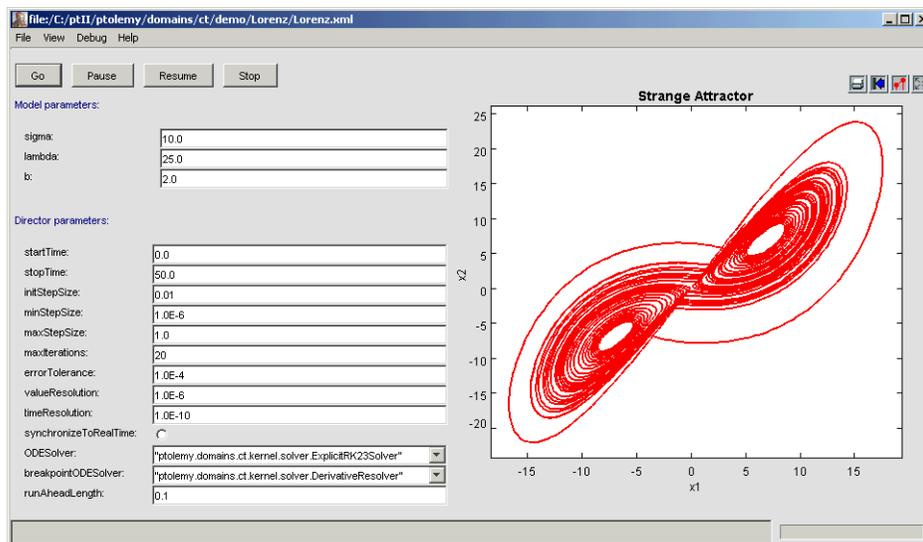
FIGURE 5. Run Window, obtained via the View menu, for the Lorenz model shown in figure 2.

dragged onto the page on the right. These are *actors* (functional blocks) and *utilities* (annotations, hierarchical models, etc.). The page on the right is almost blank, containing only a solver. The lower left corner contains a *navigation area*, which always shows the entire model (which currently consists only of a solver). For large models, the navigation area makes it easy to see where you are and makes it easy to get from one part of the model to another.

2.2.1 A Simple Sine Wave Model

We can begin by populating the model with functional blocks. Let's begin with the simple objective of generating and plotting a sine wave. There are a number of ways to do this, and the alternatives illustrate a number of interesting features about HyVisual. Open the *actor library* in the palette, and drag in the *ContinuousSinewave* actor from the *timed sources* library and the *TimedPlotter* from the *timed sinks* library. Connect the output of the *ContinuousSinewave* to the input of the *TimedPlotter* by dragging from one port to the other. The result should look something like figure 7.

The model is ready to execute. To execute it, click on the run button in the toolbar, or invoke the Run Window from the view menu. The result of the run should look like figure 8. You can zoom in on the plot by clicking and dragging in the plot window. You can also customize the plot using the buttons at the upper right.

If we zoom in on the plot, turn on stems, and set the marks to "dots," then we can make the plot look like figure 9. In this figure you can see that the sine wave is hardly smooth, and that rather few samples are produced by the simulation. It is worth understanding why this is. Consider the solver parameters shown in figure 3. Notice that the *initStepSize* parameter has value 0.1, which is coincidentally the spacing between samples in figure 9. The spacing between samples is called the *step size* of the solver. If you change *initStepSize* to 0.01 (by double clicking on the solver) and re-run the simulation, then the same region of the plot looks like figure 10. The spacing between samples is now 0.01.
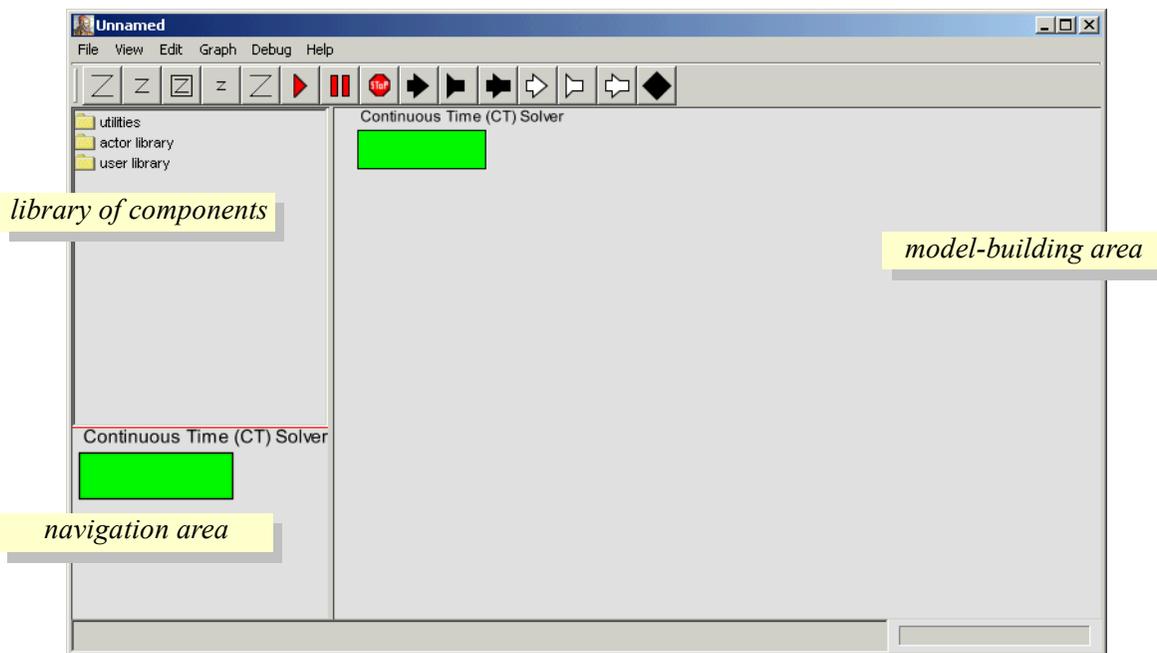
FIGURE 6. A blank model, obtained via File, New, and Graph Editor in the menus.

The model shown in figure 7 is atypical of continuous-time models of dynamical systems. It has no blocks that control the step size. Such blocks include those from the *dynamics* and *to discrete* library. For example, another way to get the sine wave to be sampled with a sampling interval of 0.01 is shown in figure 11. The *PeriodicSampler* block has a parameter *samplePeriod* that you can set to 0.01 (by double clicking on the block). This will result in the same plot as shown in figure 10, irrespective of the *initStepSize* parameter of the solver.

The models shown in figures 7 and 11 have no blocks from the *dynamics* library, and hence do not immediately represent an ordinary differential equation. When blocks from the *dynamics* library are used, then the solver uses sophisticated techniques to determine the spacing between samples. The ini-
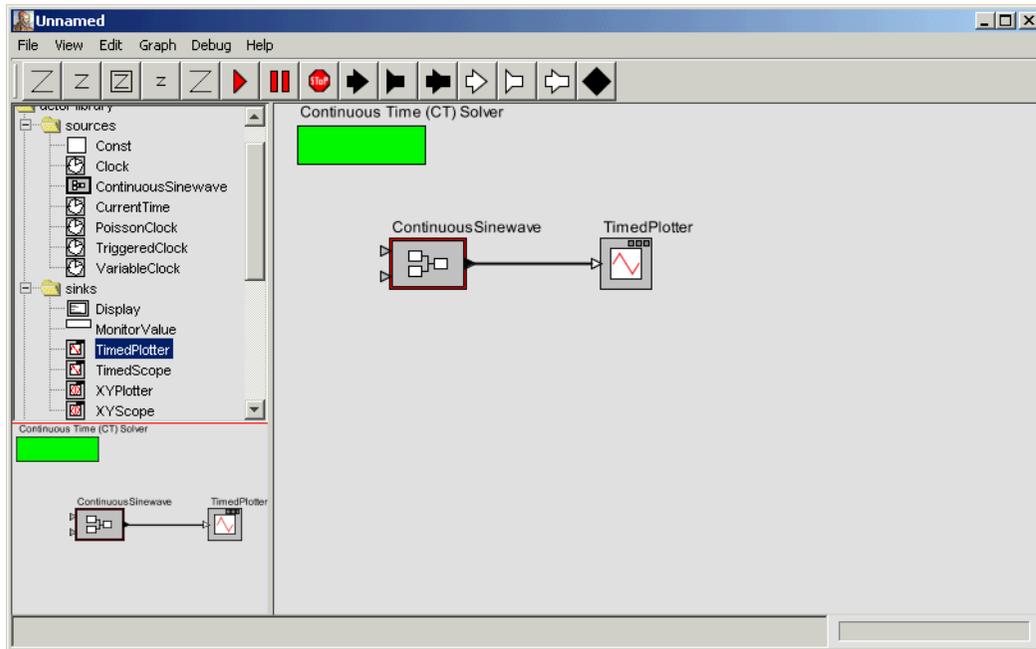
FIGURE 7. A model populated with two actors.
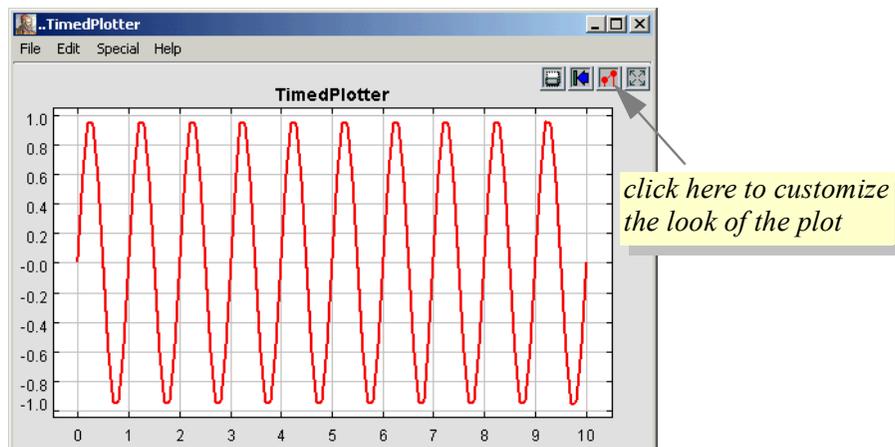
FIGURE 8. Execution of the sine wave example in figure 7, where all parameter values have default.values.

tial step size is given by *initStepSize*, but the solver may adjust it to any value between *minStepSize* and *maxStepSize*. In the case of the model in figure 7, there are no blocks with continuous dynamics, and no other blocks that affect the step size and hence there is no basis for the solver to change the step size. Thus, the step size remains at the value given by *initStepSize* for the duration of the simulation.
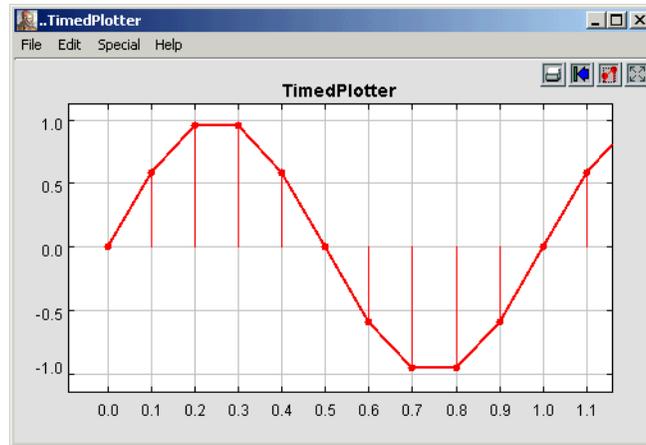
FIGURE 9. Zoomed version of the plot in figure 8, with "dots" and "stems" turned on.
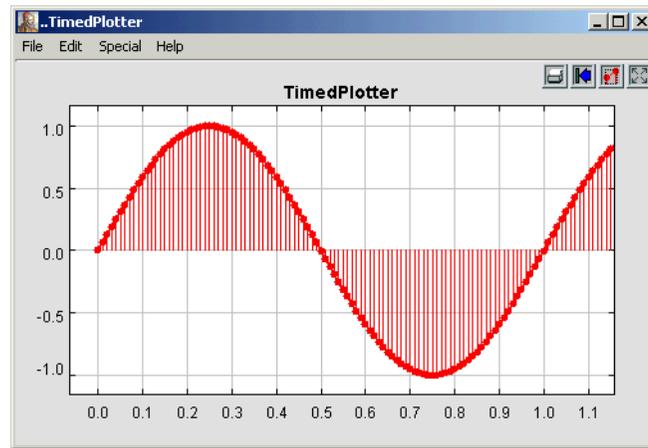
FIGURE 10. The result of running the model in figure 7 with the *initStepSize* parameter of the solver being 0.01.
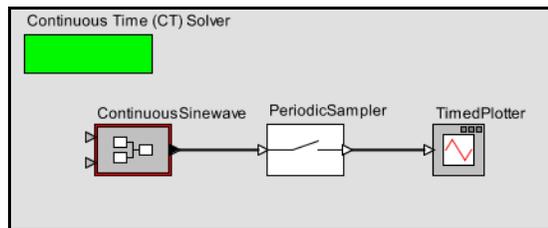
FIGURE 11. Another way to control the step size is to insert a sampler.

We will next modify the model to be more typical by describing an ODE whose solution is a sine wave. Before we do that, however, you may want to explore certain features of the user interface:

- You can save your model using commands in the File menu. File names for Ptolemy II models should end in ".xml" or ".moml" so that Vergil will properly process the file the next time you open that file.

- You can obtain documentation for the solver, or any other block in the system, by right clicking on it to get a context menu, and selecting "Get Documentation."

- You can move blocks around by clicking on them and dragging. Connections are preserved.

- You can edit the parameters of any block (including the solver) by either double clicking on it, or right clicking and selecting "Configure."

- You can change the name of a block (or even hide it) by right clicking on the block and selecting "Customize Name."

- If your installation includes the source code, then you can examine the source code for any block by right clicking and choosing "Look Inside."

2.2.2 A Dynamical System Producing a Sine Wave

From the theory of continuous-time dynamical systems, we know that an LTI system with poles on the imaginary axis will produce a sinusoidal output. That is, a system with transfer function of the form

$$H(s) = \frac{\omega_0}{(s - j\omega_0)(s + j\omega_0)} = \frac{\omega_0}{s^2 + \omega_0^2} \tag{4}$$

has an impulse response

$$h(t) = \sin(\omega_0 t)u(t), \tag{5}$$

where $u(t)$ is the unit step function. If the input to this system is a continuous-time signal x and the output is y, then the relationship between the input and output can be described by the differential equation

$$\omega_0 x(t) = \omega_0^2 y(t) + \ddot{y}(t), \tag{6}$$

where $\ddot{y}$ is the second derivative of y. Suppose that the input is zero for all time,

$$\forall t \in \Re, x(t) = 0. \tag{7}$$

Then the output satisfies

$$\ddot{y}(t) = -\omega_0^2 y(t). \tag{8}$$

This output can be generated by the model shown in figure 12. As shown in the annotations in the figure, $\ddot{y}$ is calculated by multiplying y by $-\omega_0$, y is calculated by integrating $\dot{y}$, and $\dot{y}$ is calculated by integrating $\ddot{y}$. If we set the initial state of the left integrator to 1.0 and run the model for 5 time units, we get the result shown in figure 13.

The model in figure 12 shows two additional key features of the user interface, the mechanism for connecting an output to multiple inputs (relations) and the mechanism for defining and using parameters. We discuss these two mechanisms next.

2.2.3 Making Connections

The models in figures 7 and 11 have simple connections between blocks. These connections are made by clicking on one port and dragging to the other. The connection is maintained if either block is moved. We can now explore how to create and manipulate more complicated connections, such as the ones in figure 12, where the output of the right *Integrator* goes to both the *Scale* and the *TimedPlotter* blocks. Such connections are mediated by a *relation*, indicated by a black diamond, as shown in figure 12. A relation can be linked to one output port and any number of input ports.

If we simply attempt to make the connections by clicking and dragging from the *Integrator* output port to the two input ports in sequence, then we get the exception shown in figure 14. Such exceptions can be intimidating, but are the normal and common way of reporting errors in HyVisual. The key line in this exception report is the last one, which says

```
Attempt to link more than one relation to a single port.
```
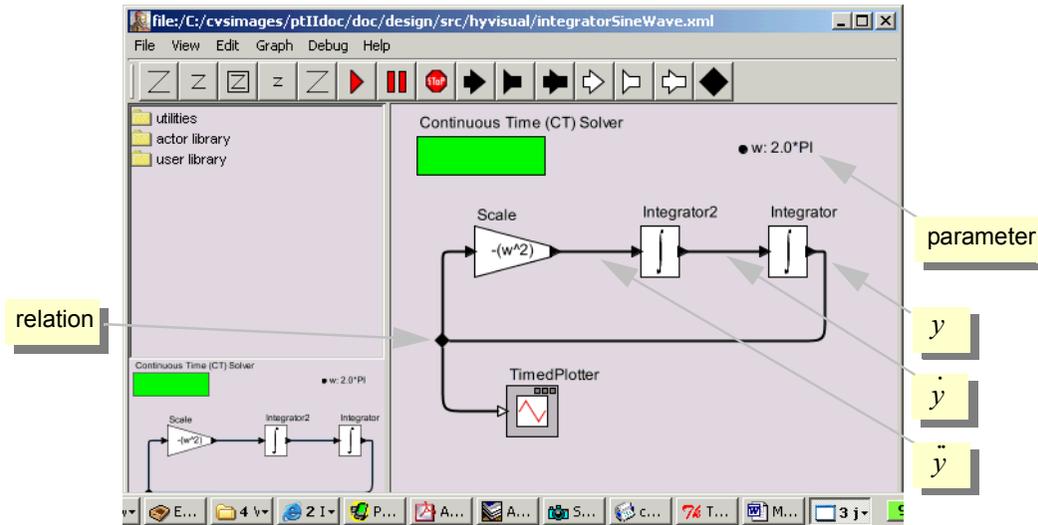

FIGURE 12. Model that generates a sine wave if the integrators have a non-zero initial condition.
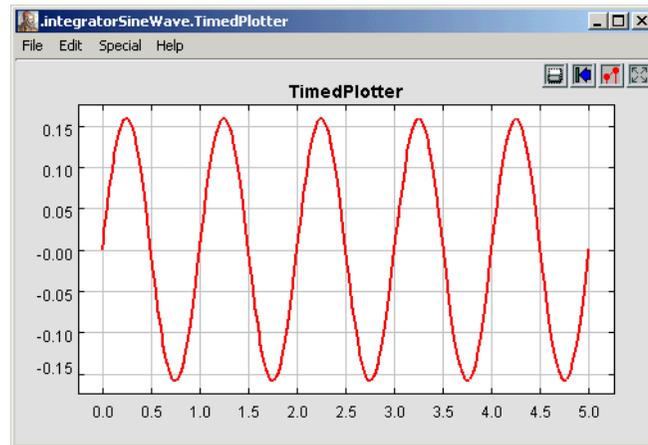
FIGURE 13. Result of running the model in figure 12 with the *initialState* of the left integrator set to 1.0.

The line below that gives the names of the objects involved, which are

```
in .integratorSineWave.Integrator.output and .integratorSineWave.relation4
```

In HyVisual models, all objects have a dotted name. The dots separate elements in the hierarchy. Thus, ".integratorSineWave.Integrator.output" is an object named "output" contained by an object named "Integrator", which is contained by a model named "integratorSineWave."

Why did this exception occur? The diagram shows two distinct flavors of ports, indicated in the diagrams by a filled triangle or an unfilled triangle. The output port of the *Integrator* block is a *single port*, indicated by a filled triangle, which means that it can only support a single connection. The input port of the *TimedPlotter* block is a *multiport*, indicated by unfilled triangles. Multiports can support multiple connections, where each connection is treated as a separate *channel*. A channel is a path from an output port to an input port (via relations) that can transport a single stream of tokens.

So how do we get the output of the *Integrator* to the other two actors? We need an explicit *relation* in the diagram. A relation is represented in the diagram by a black diamond, as shown in Figure 15. It can be created by either control-clicking on the background or by clicking on the button in the toolbar with the black diamond on it.

Making a connection to a relation can be tricky, since if you just click and drag on the relation, the relation gets selected and moved. To make a connection, hold the control button while clicking and dragging on the relation.
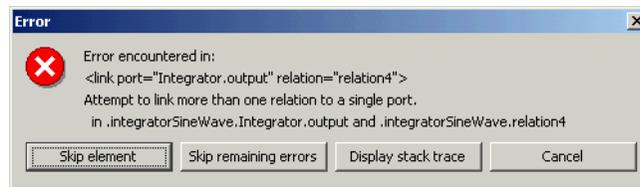
FIGURE 14. An exception that results from attempting to make a multi-way connection without a relation.
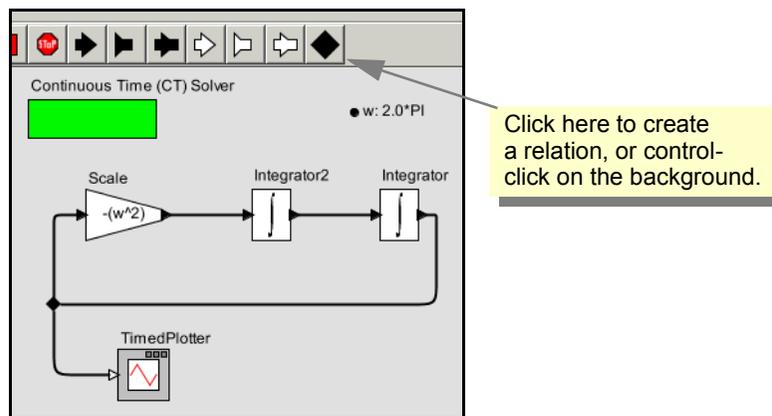
FIGURE 15. A relation can be used to broadcast an output from a single port.

In the model shown in figure 15, the relation is used to broadcast the output from a single port to a number of places. The single port still has only one connection to it, a connection to a relation. Relations can also be used to control the routing of wires in the diagram. For example, in figure 15, the relation is placed to the left of all the blocks in order to get a pleasing layout. However, as of this writing, a connection can only have a single relation on it, so the degree to which routing can be controlled is limited.

The *TimedPlotter* in figure 15 has a multiport input, as indicated by the unfilled triangle. This means that multiple channels of input can be connected directly to it. Consider the modification shown in figure 16, where both y and $\dot{y}$ are connected (via relations) to the input port of the *TimedPlotter*. The resulting plot is shown at the right. The two signals are treated by the block as distinct input signals coming in on separate channels.

2.2.4 Parameters

Figure 16 shows a parameter named "w" with value "2.0*PI." That parameter is then used in the *Scale* block to specify that the scale factor is "−(w^2)." This usage illustrates a number of convenient features.

To create a parameter that is visible in the diagram, drag one in from the *utilities* library, as shown in figure 17. Right click on the parameter to change its name to "w". (In order to be able to use this parameter in expressions, the name cannot have any spaces in it.) Also, right click or double click on the parameter to change its default value to "2.0*PI." This is an example of the sort of expressions you can use to define parameter values. The expression language is described below in section 5.

The parameter, once created, belongs to the model. If you right click on the grey background and select Configure, then you can edit the parameter value, just as you could by double clicking on the parameter. The resulting dialog also allows you to create parameters that are not visible in the model. The parameter is also visible and editable in the Run Window obtained via the View menu.

A parameter of the model can be used in expressions anywhere in the model, including in parameter values for blocks within the model. In figure 16, for instance, the *factor* parameter of the *Scale* actor has the value "−(w^2)," which references the parameter *w*.

2.2.5 Annotations

There are several other useful enhancements you could make to this model. Try dragging an *annotation* from the *utilities* library and creating a title on the diagram. Also, try setting the title of the plot
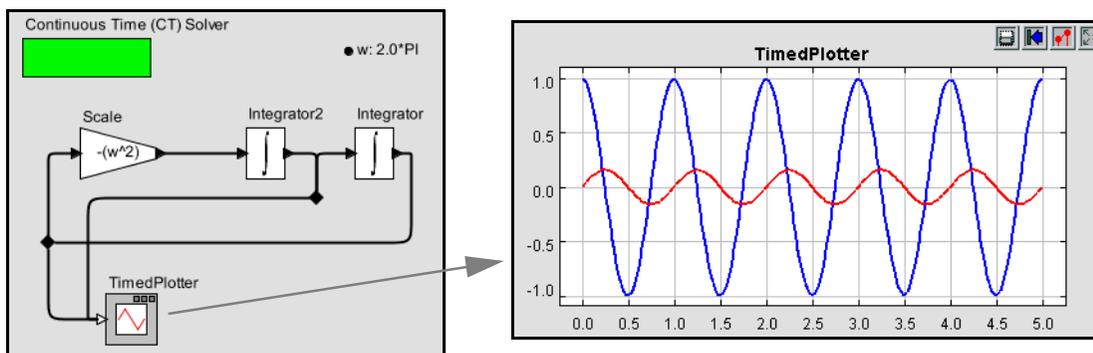
FIGURE 16. Multiple signals can be sent to a multiport, shown with the unfilled triangle on the *TimedPlotter*. In this case, two signals are plotted.

by clicking on the second button from the right in the row of buttons at the top right of the plot. This button has the tool tip "Set the plot format" and brings up the format control window.

2.2.6 Impulse Response

Consider equation (6), which we repeat here for reference:

$$\omega_0 x(t) \;=\; \omega_0^2 y(t) + \ddot{y}(t).\tag{9}$$

Figure 12 and subsequent models based on this equation assume that the input is zero for all time, $x(t) = 0$, thus realizing equation (8). We can elaborate on this model by re-introducing the input, and allowing it to be non-zero. The result is shown in figure 18, where the input is provided by a block labeled "Clock." Notice that the input is multiplied by ω_0, and then $\omega_0^2 y(t)$ is subtracted from it to obtain $\ddot{y}(t)$. I.e., it calculates

$$\ddot{y}(t) \;=\; \omega_0 x(t) - \omega_0^2 y(t).\tag{10}$$
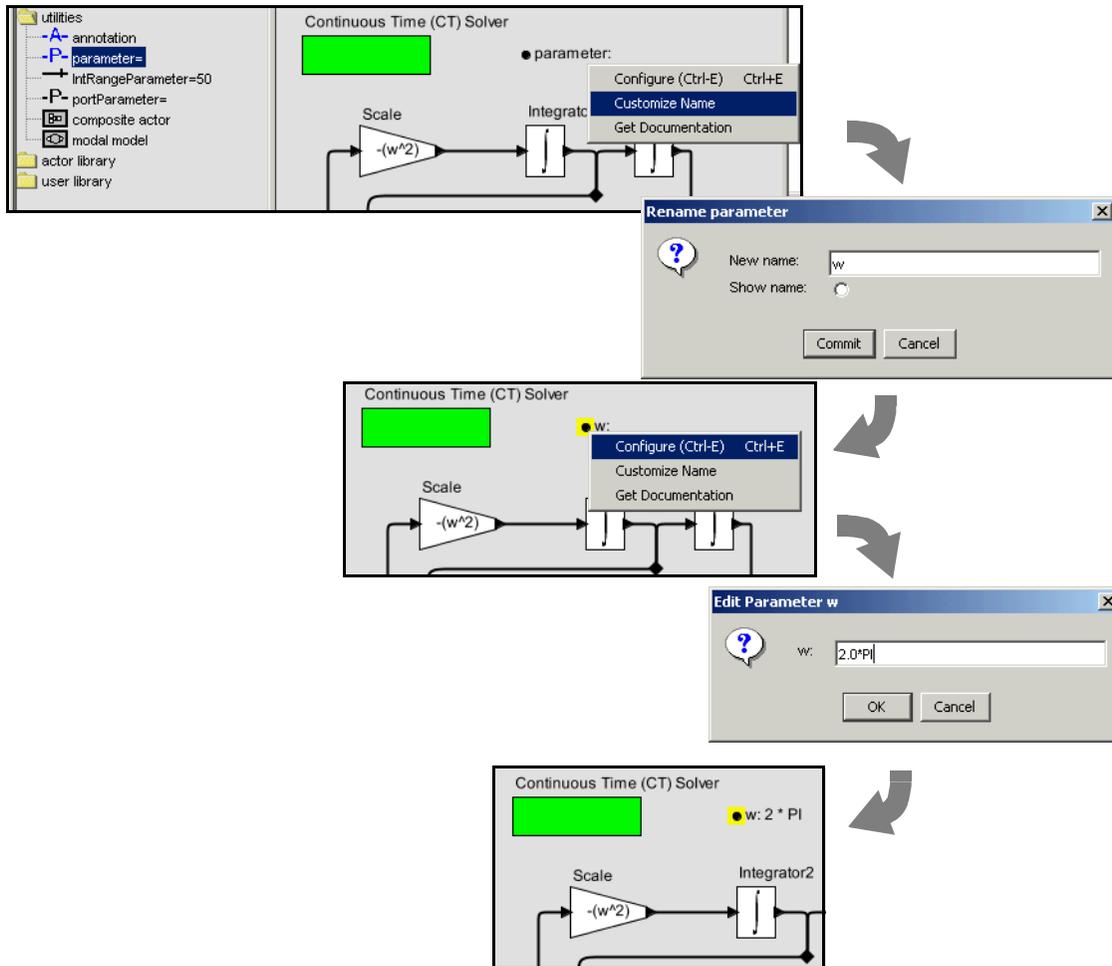
FIGURE 17. Adding a parameter to the channel model.

In the model, both integrators now have *initialState* set to 0.0. The plot in the figure shows the result of running the model with an impulse as the input, resulting in an impulse response that matches (5).

The key question, however, is how can we generate an impulse for the input? In theory, an impulse, also known as a Dirac delta function, is a continuous-time function δ that satisfies

$$\delta(t) = 0, \forall t \neq 0$$

$$\int_{-\infty}^{\infty} \delta(t)dt = 1 \quad . \tag{11}$$

From these relations, it is easy to see that the Dirac delta function must have infinite value at $t = 0$, because otherwise it could not integrate to one. Hence, it is problematic to generate an impulse in a continuous-time simulator.

The model shown in figure 18 uses a Clock actor to generate an approximate impulse. The parameters of the Clock actor are shown in figure 19. First, notice that *numberOfCycles* is set to 1. This means that only one pulse will be generated. The pulse is defined by the *offsets* and *values* parameters. The *offsets* parameter is set to {0.0, 1.0E-5}, which is an array with two numbers. The *values* parameter is set to {1.0E5, 0.0}. Together, these mean that the output goes to value 1.0×10^5 at time 0.0, and
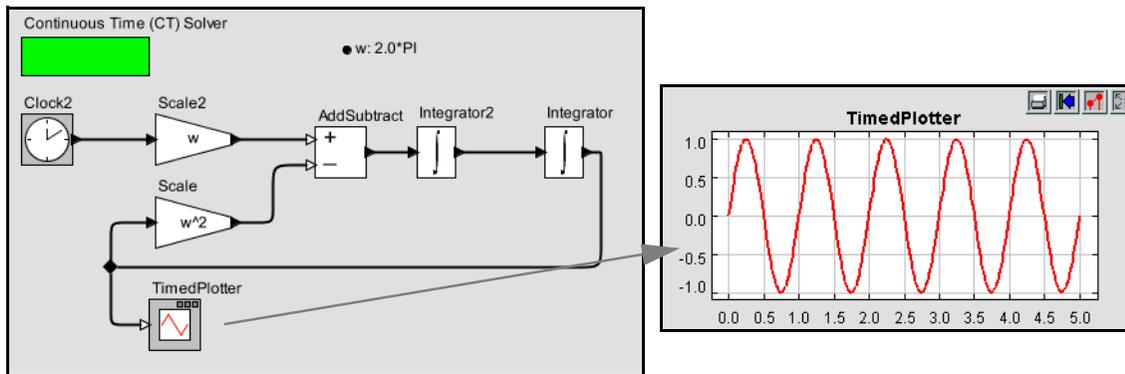
FIGURE 18. Variant of figure 12 that has an input, which is provided by the Clock actor.
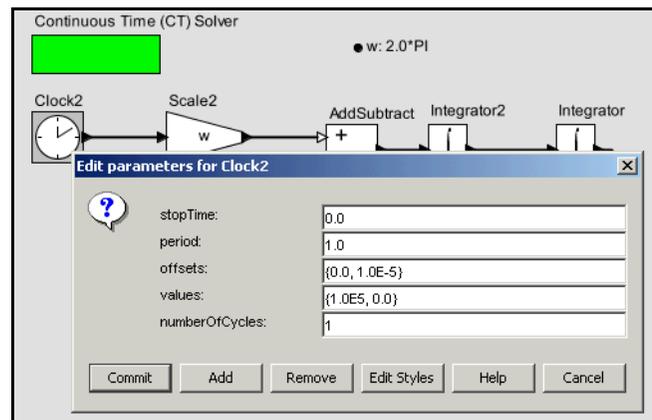
FIGURE 19. Parameters of the Clock actor that get it to output an approximate impulse.

then to value 0.0 at time 1.0×10^{-5}. Thus, the output is a very narrow, very high rectangular pulse, with unit area.

If you create a Clock and set these parameter values, and try to run the model, you are likely to see the exception shown in figure 20. The problem here is that the default *minStepSize* value for the solver, as shown in figure 3, is too large, given the very narrow pulse we are trying to generate. In this case, it is sufficient to change the *minStepSize* parameter to 1.0E-9. Generally, the *minStepSize* parameter needs to be considerably smaller than the smallest phenomenon in time that is being observed. It is worth noting that even with this small value for *minStepSize*, the solver does not actually use step sizes anywhere near this very often. You can examine which points in the output plot are actually computed by the solver by turning on stems in the output plot.

2.2.7 Using Higher-Order Dynamics Blocks

Recall from (4) that a transfer function given by the Laplace transform

$$H(s) = \frac{\omega_0}{s^2 + \omega_0^2} \tag{12}$$

describes the system shown in figure 18. In fact, we could have constructed the system more easily by using the ContinuousTransferFunction actor in the *dynamics* library, as shown in figure 21. That actor has as parameters two arrays, *numerator* and *denominator*, which are set to {w} and {1.0, 0.0, w^2}, respectively. A portion of the documentation for that actor is shown in figure 22 (you can obtain this documentation by right clicking on the actor icon and selecting Get Documentation). As indicated on that page, the *numerator* and *denominator* parameters give the coefficients of the numerator and denominator polynomials in (12).

Recall that to run this model, you will need to set the *minStepSize* parameter of the solver to 10^{-9} or smaller.
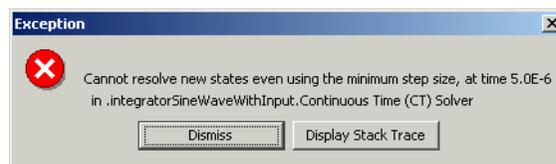
FIGURE 20. Exception due to running the model with the *minStepSize* parameter set too high.
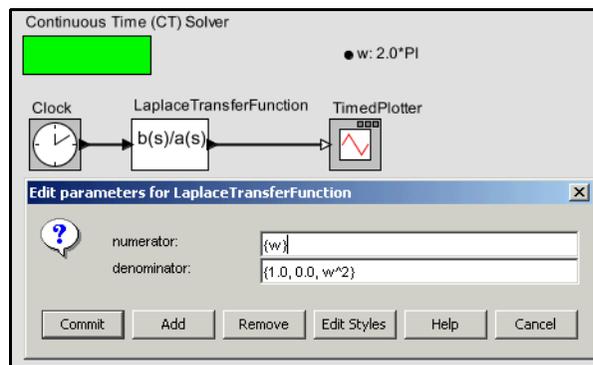
FIGURE 21. A model equivalent to that in figure 18, but using the ContinuousTransferFunction actor.

An interesting curiosity about this actor is how it works. It works by creating a hierarchical model similar to the one that we built by hand. If, after running the model at least once, you right click on the ContinuousTransferFunction icon and select Look Inside (or type Control-L over the icon), you will see an inside model that looks like that shown in figure 23. This model is hard to interpret, since all the icons are placed one on top of the other at the upper left. You can select Automatic Layout from the Graph menu to get something a bit easier to read, shown in figure 24. It is still far from perfect, but with a bit of additional placement effort, you can verify that this model is functionally equivalent to the one we constructed manually in figure 18.

FIGURE 22. A portion of the documentation for the ContinuousTransferFunction actor.
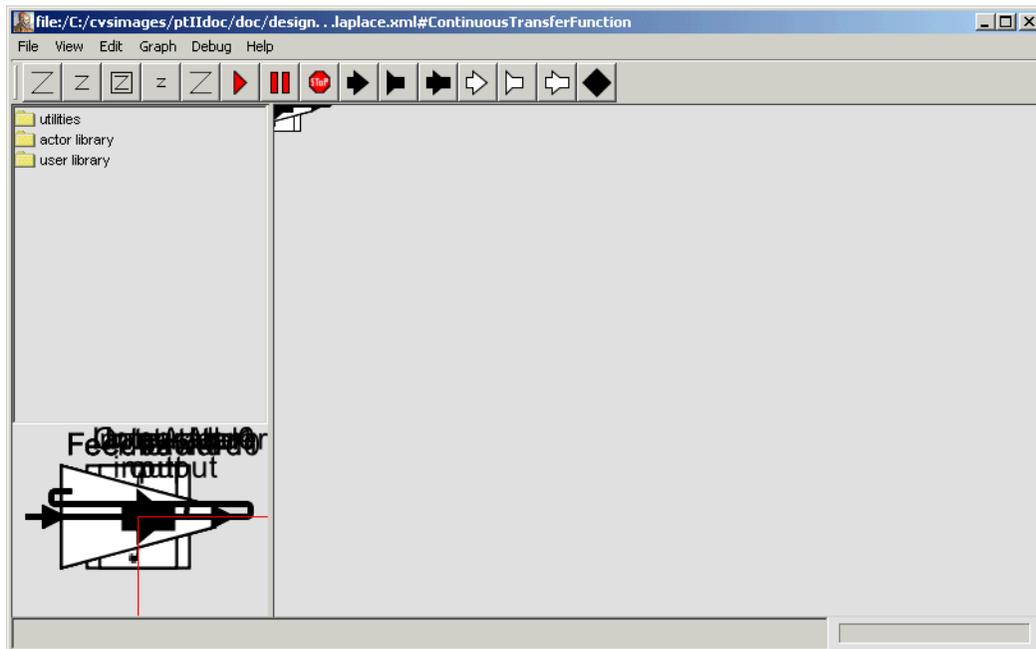
FIGURE 23. Inside the ContinuousTransferFunction actor of figure 21.

2.3 Data Types

In the example of figure 7, the *ContinuousSinewave* actor produces values on its output port. The values in this case are *double*. You can examine the data types of ports after executing a model by simply lingering on the port with the mouse. A tooltip will appear, as shown in figure 25. Most actors in the actor library are *polymorphic*, meaning that they can operate on or produce data with multiple types. The behavior may even be different for different types. Multiplying matrices, for example, is not the same as multiplying integers, but both are accomplished by the *MultiplyDivide* actor in the *math library*. HyVisual includes a sophisticated type system[1] that allows this to be done efficiently and safely. Actors represent type constraints that relate the types of the their ports and parameters, and the type system resolves the constraints, unless a conflict arises.

To explore data types a bit further, try creating the model in Figure 26. The *Const* and *CurrentTime* actors are listed under *timed sources* within *sources*, the *AddSubtract* actor is listed under *math,* and the *MonitorValue* actor is listed under *timed sinks* within *sinks*. Set the *value* parameter of the constant to be 1. Running the model for 10.0 time units should result in 9.0 being displayed in *MonitorValue*, as
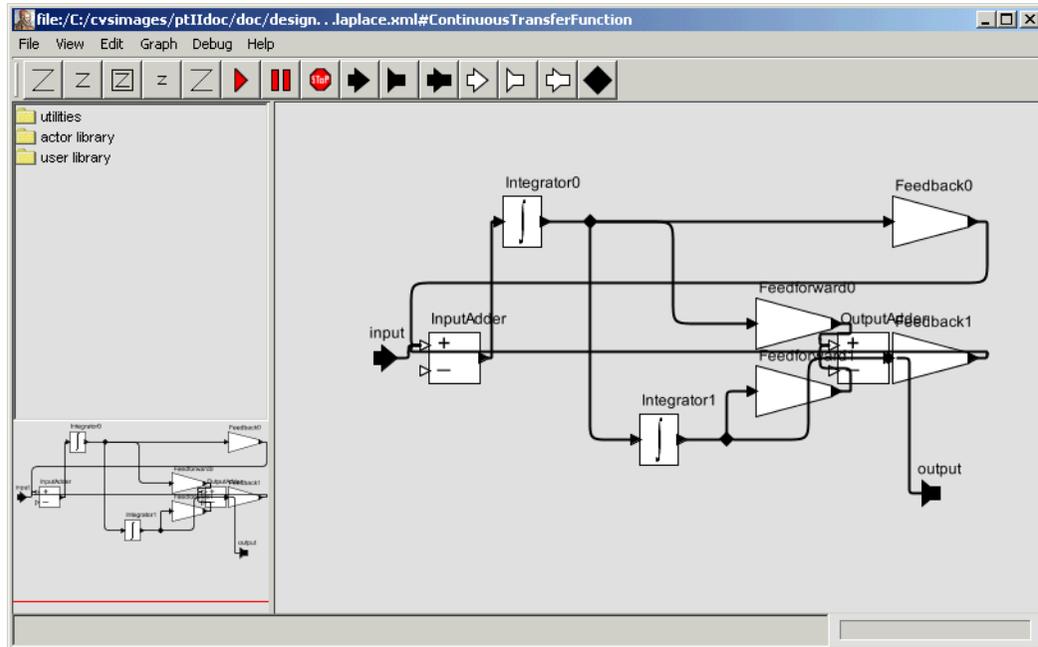
FIGURE 24. The diagram of figure 23, after invoking Automatic Layout from the Graph menu.
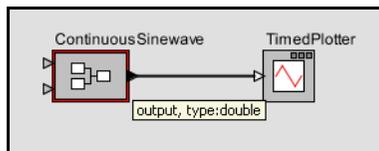
FIGURE 25. Tooltip showing the name and data type of the output port of the ContinuousSinewave of figure 7.

---

1. Developed by Yuhong Xiong and realized in Ptolemy II.

shown in the figure. The output of the *CurrentTime* actor is a *double*, the output of the *Const* actor is an *int*, and the *AddSubtract* actor adds these two to get a *double*.

Now for the real test: change the value of the *Const* actor to a string, such as "a" (with the quotation marks). In fact, the *Const* actor can have as its *value* anything that can be given using the expression language, explained below in section 5. When you execute the model, you should see an exception window, as shown in Figure 27. Do not worry; exceptions are a normal part of constructing (and debugging) models. In this case, the exception window is telling you that you have tried to subtract a *string* value from an *double* value, which doesn't make much sense at all (following Java, adding strings *is* allowed). This is an example of a type error.

We can make a small change to the model to get something that does not trigger an exception. Disconnect the *Const* from the lower port of the *AddSubtract* actor and connect it instead to the upper port, as shown in Figure 28. You can do this by selecting the connection and deleting it (using the delete key), then adding a new connection, or by selecting it and dragging one of its endpoints to the new location. Notice that the upper port is an unfilled triangle; this indicates that it is a *multiport*, meaning that you can make more than one connection to it. Now when you run the model you should see strings like "10.0a", as shown in the figure. This is the result of converting the *double* from *CurrentTime* to a *string* "10.0," and then adding the strings together (which, following Java, means concatenating them).

As a rough guideline, HyVisual will perform automatic type conversions when there is no loss of information. An integer can be converted to a string, but not vice versa. An integer can be converted to a double, but not vice versa. An integer can be converted to a long, but not vice versa.
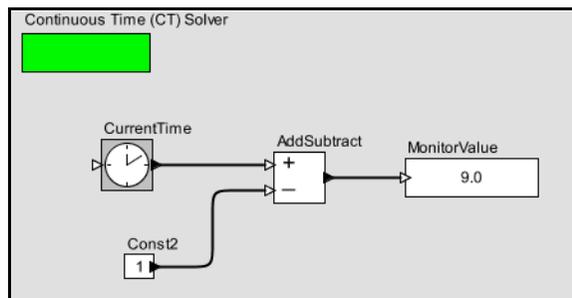
FIGURE 26. Another example, used to explore data types in HyVisual.
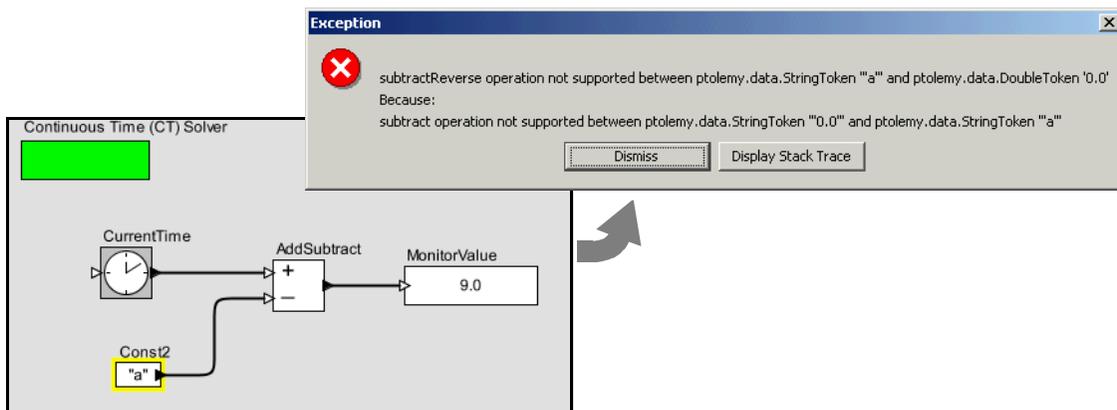
FIGURE 27. An example that triggers an exception when you attempt to execute it. Strings cannot be subtracted from doubles.

2.4 Hierarchy

HyVisual supports (and encourages) hierarchical models. These are models that contain components that are themselves models. Such components are called *composite actors*. Suppose we wish to take the sine wave generated by the previous examples and send it through a model of a noisy channel. We will create a composite actor modeling the channel, and then use that actor in a model.

2.4.1 Creating a Composite Actor

First open a new graph editor and drag in a *Typed Composite Actor* from the *utilities* library. This actor is going to add noise to our measurements. First, using the context menu (obtained by right clicking over the composite actor), select "Customize Name", and give the composite a better name, like "Channel", as shown in Figure 29. Then, using the context menu again, select "Look Inside" on the actor. You should get a blank graph editor, as shown in Figure 30. The original graph editor is still open. To see it, move the new graph editor window by dragging the title bar of the window. Notice that the new window has no solver. It will be executed by the solver belonging to the parent model.

2.4.2 Adding Ports to a Composite Actor

First we have to add some ports to the composite actor. There are several ways to do this, but clicking on the port buttons in the toolbar is probably the easiest. You can explore the ports in the toolbar by lingering with the mouse over each button in the toolbar. A tool tip pops up that explains the button.
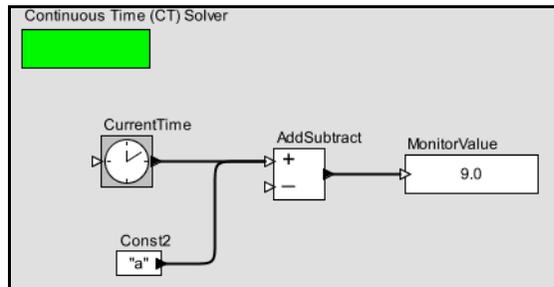
FIGURE 28. Addition of a string to an integer.
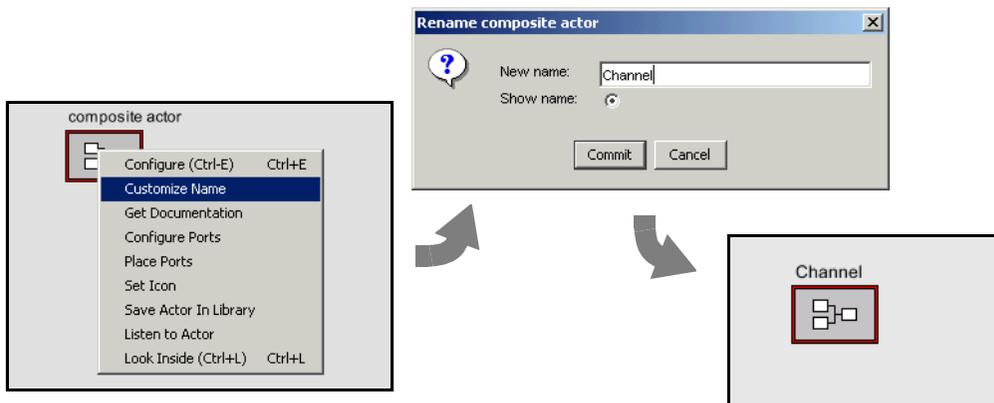
FIGURE 29. Changing the name of an actor.

The buttons are summarized in Figure 31. Create an input port and an output port and rename them *input* and *output* right by clicking on the ports and selecting "Customize Name". Note that, as shown in Figure 32, you can also right click on the background of the composite actor and select *Configure Ports* to change whether a port is an input, an output, or a multiport. The resulting dialog also allows
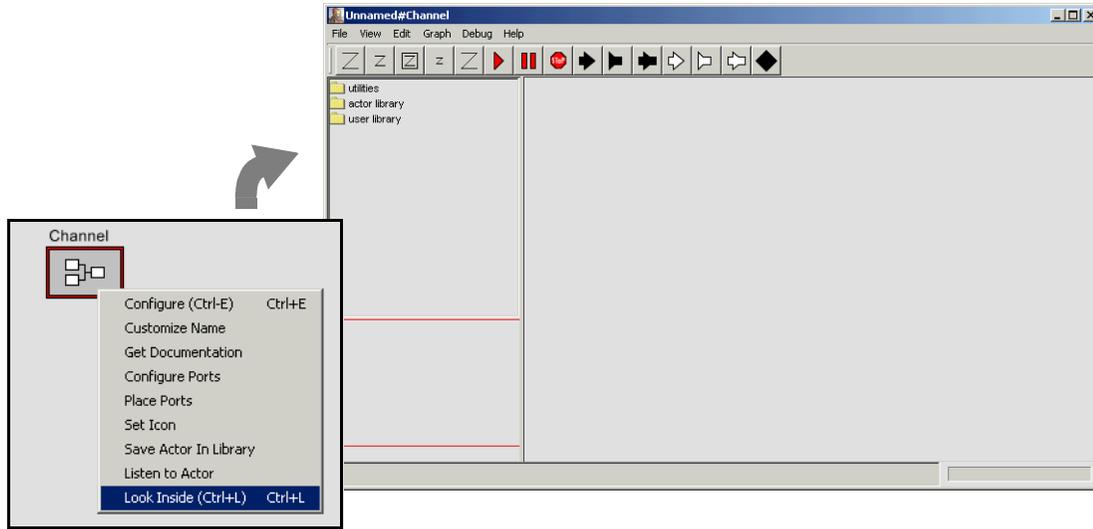
FIGURE 30. Looking inside a composite actor.
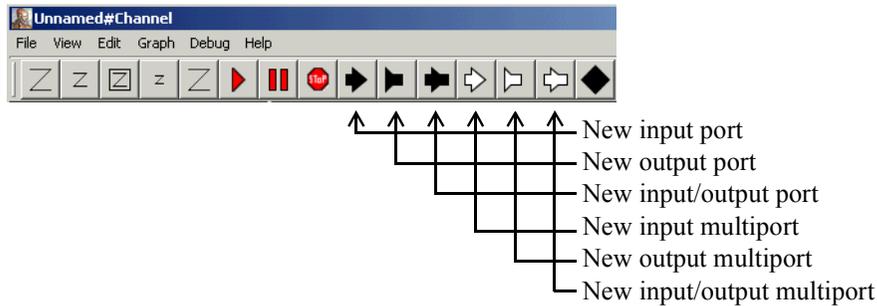
FIGURE 31. Summary of toolbar buttons for creating new ports.
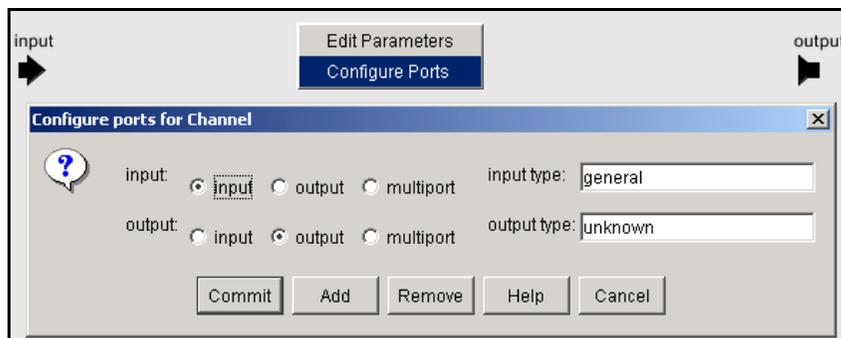
FIGURE 32. Right clicking on the background brings up a dialog that can be used to configure ports.

you to set the type of the port, although much of the time you will not need to do this, since the type inference mechanism in Ptolemy II will figure it out from the connections.

Then using these ports, create the diagram shown in Figure 33[1]. The *Gaussian* actor creates values from a Gaussian distributed random variable, and is found in the *random* library. Now if you close this editor and return to the previous one, you should be able to easily create the model shown in figure 34, similar to the model in figure 7, but with a channel. Notice that both our new *Channel* actor and the *ContinuousSinewave* actor have a red outline. The *ContinuousSinewave* actor is also a composite actor (try looking inside). If you execute this model, you should see something like the plot shown in figure 34.

2.4.3 Setting the Types of Ports

In the above example, we never needed to define the types of any ports. The types were inferred from the connections. Indeed, this is usually the case in Ptolemy II, but occasionally, you will need to set the types of the ports. Notice in Figure 32 that there is a position in the dialog box that configures ports for specifying the type. Thus, to specify that a port has type *boolean*, you could enter *boolean* into the dialog box. There are other commonly used types: *complex*, *double*, *fixedpoint*, *general*, *int*, *long*, *matrix*, *object*, *scalar*, *string*, and *unknown*. To set the type to a double matrix, say:

```
[double]
```

This expression actually creates a 1 by 1 matrix containing a double (the value of which is irrelevant). It thus serves as a prototype to specify a double matrix type. Similarly, we can specify an array of complex numbers as
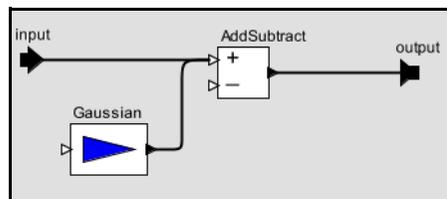
FIGURE 33. A simple channel model defined as a composite actor.
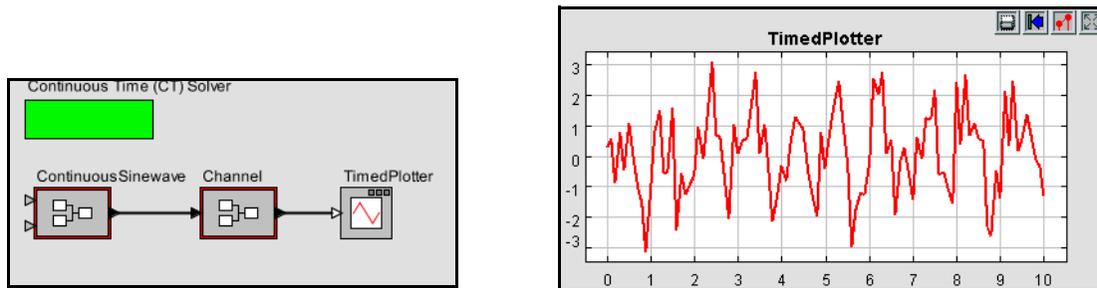
FIGURE 34. A model producing a noisy sine wave.

1. **Hint:** to create a connection starting on one of the external ports, hold down the control key when dragging.

```
{complex}
```

In the Ptolemy II expression language, square braces are used for matrices, and curly braces are used for arrays. To specify a record containing a string named "name" and an integer named "address," say:

```
{name=string, address=int}
```

2.5 Discrete Signals and Mixed-Signal Models

Continuous-time signals can be combined with discrete-event signals in the same model. A discrete signal in HyVisual is one that consists of *events* that are placed on the time line and have a value. If a discrete signal is examined at a time where there is no event, then the signal will have no value. For some actors, this is an error. Continuous actors cannot read discrete signals. Most actors can handle either signal, however, so usually it does not require much effort to mix the two types of signals. For example, most of the math actors are equally content working on continuous signals as discrete signals.

Discrete signals are generated by the actors in the *to discrete* library. These include an actor to simply periodically generate discrete events, level-crossing detectors, samplers (like that shown in figure 11), and a threshold monitor. Discrete signals are converted to continuous signals by the actors in the *to continuous* library. These include a zero-order hold and a first-order hold.

Usually, HyVisual will infer automatically whether a signal is discrete or continuous, but occasionally, you will need to help it. Whether a signal is *DISCRETE* or *CONTINUOUS* is referred to as its *signal type* (do not confuse this with the data type of the signal, which indicates, for example, whether it's a *double* or an *int*).

Some actors declare the signal types of their ports. For example, an *Integrator* has a CONTINUOUS input and a CONTINUOUS output; a *PeriodicSampler* has a CONTINUOUS input and a DISCRETE output; a *TriggeredSampler* has one CONTINUOUS input (the *input*), one DISCRETE input (the *trigger*), and a DISCRETE output; and a *ZeroOrderHold* has a DISCRETE input and a CONTINUOUS output.

Certain other actors declare that they operate only on sequences of data tokens (they declare this by implementing the *SequenceActor* interface). Their inputs and outputs are treated as DISCRETE. Unless otherwise specified, the types of the input ports and output ports of an actor are the same.

Sometimes, conflicts arise, and you will need to force a port to be either discrete or continuous. To do this, add a parameter named "signalType" to the port. The signal type system will recognize this parameter (by name) and resolve other types accordingly. To add this parameter, right click on the port, select Configure, then click on Add. Give the parameter as a value either the string "CONTINUOUS" or "DISCRETE", including the quotation marks.

Signal types can become an issue particularly at the boundaries of state or transition refinements, which occur in Hybrid systems, as explained below.

2.6 Navigating Larger Models

Sometimes, a model gets large enough that it is not convenient to view it all at once. There are four toolbar buttons, shown in Figure 2.27 that help. These buttons permit zooming in and out. The "Zoom

reset" button restores the zoom factor to the "normal" one, and the "Zoom fit" calculates the zoom factor so that the entire model is visible in the editor window.

In addition, it is possible to pan over a model. Consider the window shown in figure 36. Here, we have zoomed in on the Lorenz attractor model of figure 2 so that icons are larger than the default. The *pan window* at the lower left shows the entire model, with a red box showing the visible portion of the model. By clicking and dragging in the pan window, it is easy to navigate around the entire model. Clicking on the "Zoom fit" button in the toolbar results in the editor area showing the entire model, just as the pan window does.

3. Hybrid Systems
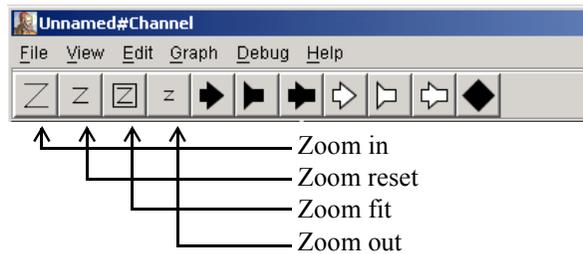
FIGURE 35. Summary of toolbar buttons for zooming and fitting.
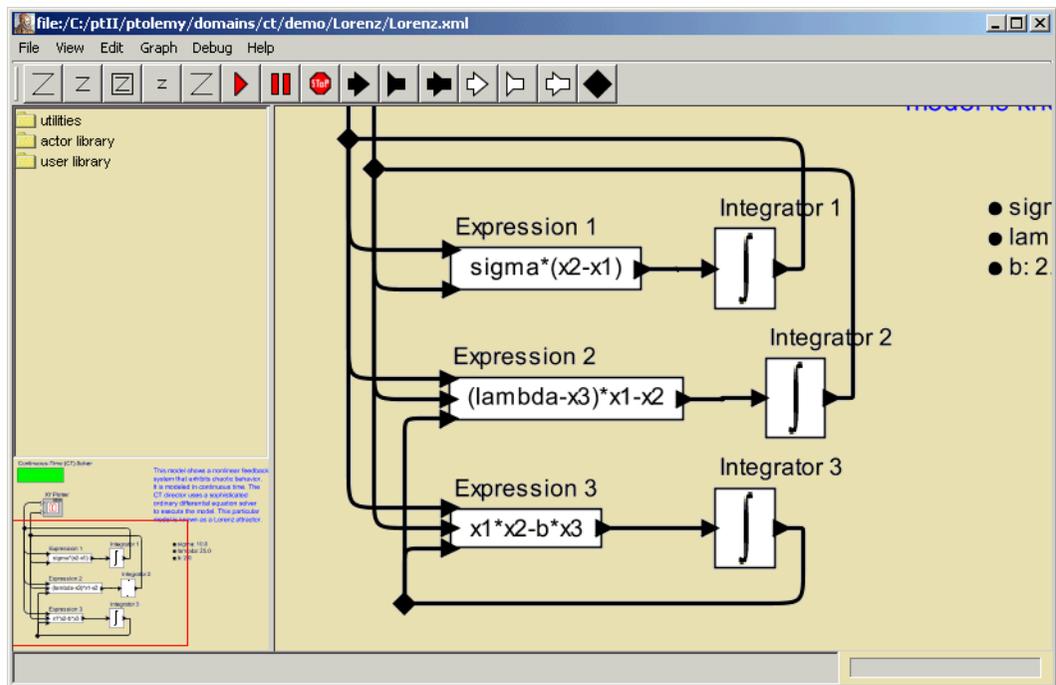
FIGURE 36. The pan window at the lower left has a red box representing the visible are of the model in the main editor window. This red box can be moved around to view different parts of the model.

Hybrid systems are models that combine continuous dynamics with discrete mode changes. They are created in HyVisual by creating a *modal model*, found in the *utilities* library. We start by examining a pre-built modal model, and conclude by illustrating how to construct one.

3.1 Examining a Pre-Built Model

The third example in figure 1 is a simple hybrid system model of a bouncing ball. The top-level contents of this model is shown in figure 37. It contains only two actors, a *Ball Model* and a *TimedPlotter*. The *Ball Model* is an instance of the *modal model* found in the *utilities* library, but renamed. If you execute the model, you should see a plot like that in the figure. The continuous dynamics correspond to the times when the ball is in the air, and the discrete events correspond to the times when the ball hits the surface and bounces.

If you look inside the *Ball Model*, you will see something like figure 38. Figure 38 shows a state-machine editor, which has a slightly different toolbar and a significantly different library at the left. The circles in figure 38 are states, and the arcs between circles are *transitions* between states. A modal model is one that has *modes*, which represent regimes of operation. Each mode in a modal model is represented by a state in a finite-state machine.

The state machine in figure 38 has three states, named *init*, *free*, and *stop*. The *init* state is the initial state, which is set as shown in figure 39. The *free* state represents the mode of operation where the ball is in free fall, and the *stop* state represents the mode where the ball has stopped bouncing.

At any time during the execution of the model, the modal model is in one of these three states. When the model begins executing, it is in the *init* state. During the time a modal model is in a state, the behavior of the modal model is specified by the *refinement* of the state. The refinement can be examined by looking inside the state. As shown in figure 40, the *init* state has no refinement.

Consider the transition from *init* to *free*. It is labeled as follows:

```
true
free.initialPosition = initialPosition; free.initialVelocity = 0.0
```
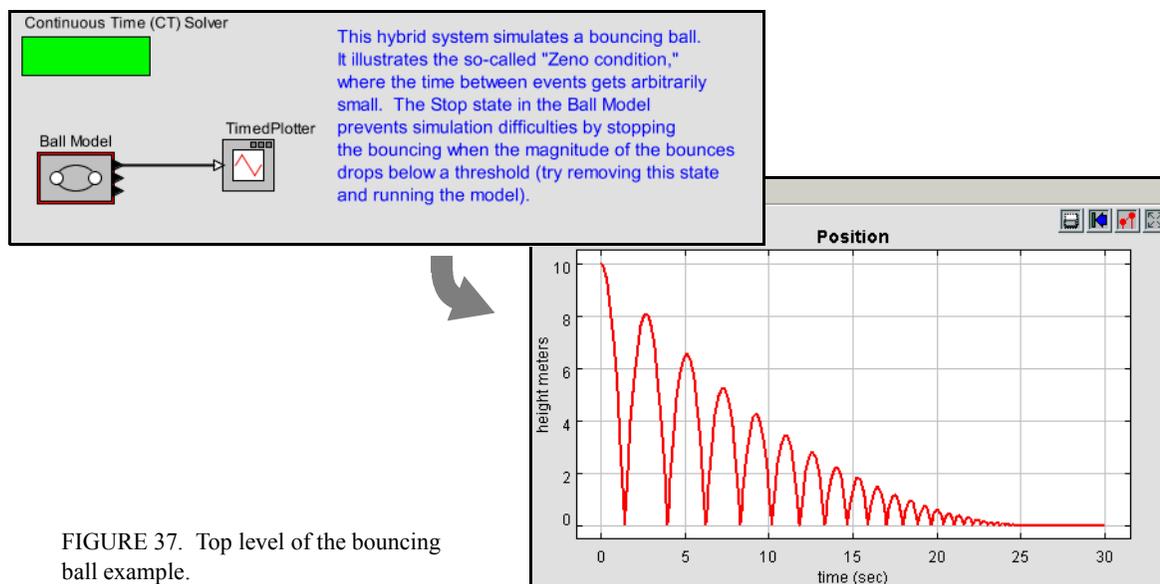

FIGURE 37. Top level of the bouncing ball example.

The first line is a *guard*, which is predicate that determines when the transition is enabled. In this case, the transition is always enabled, since the predicate has value *true*. Thus, the first thing this model will do is take this transition and change modes to *free*. The second line specifies a sequence of *actions*, which in this case set parameters of the destination mode *free*.

If you look inside the *free* state, you will see the refinement shown in figure 41. This model represents the laws of gravity, which state that an object of any mass will have an acceleration of roughly
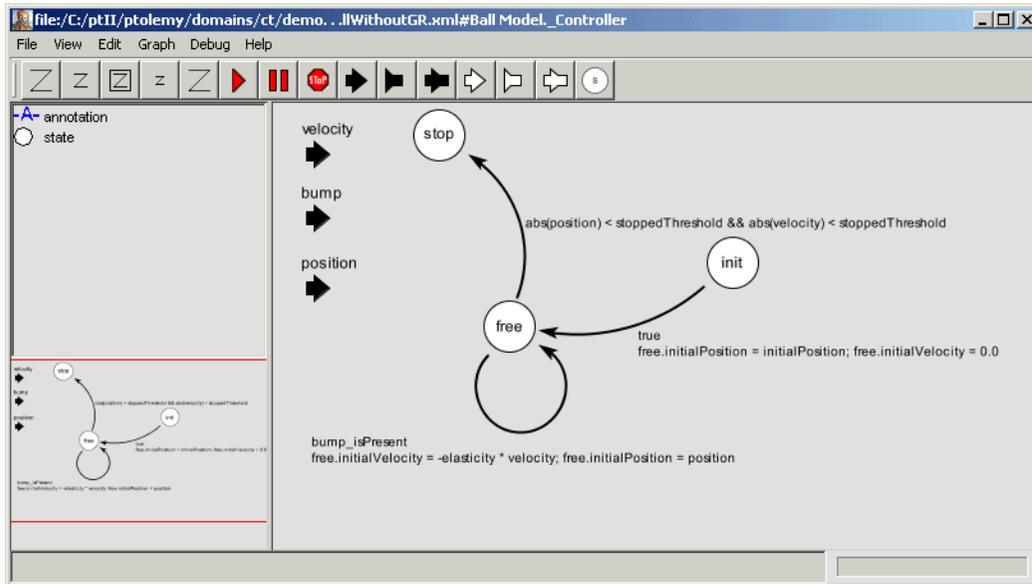
FIGURE 38. Inside the *Ball Model* of figure 37.
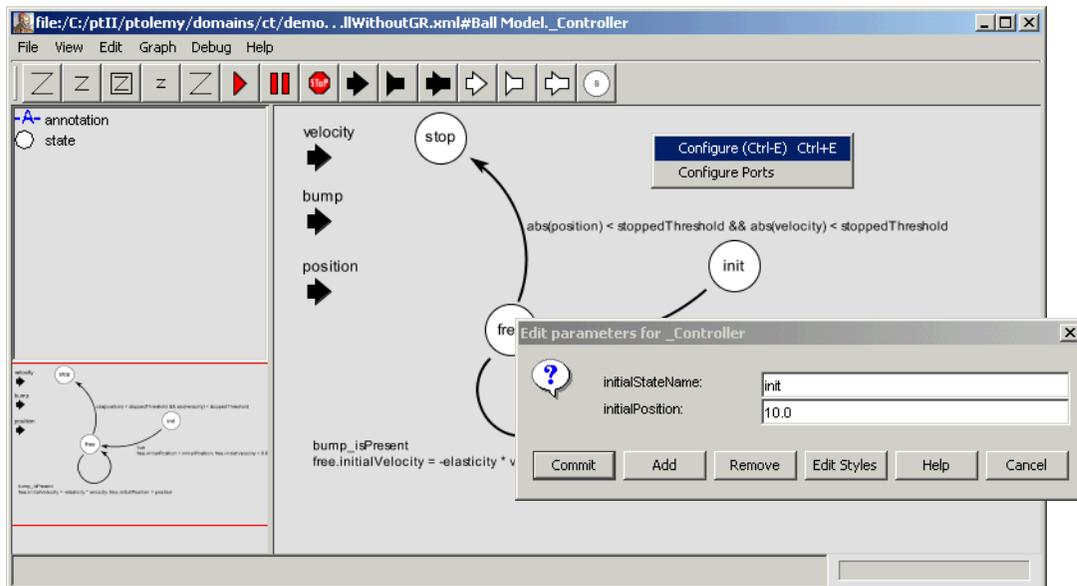
FIGURE 39. The initial state of a state machine is set by right clicking on the background and specifying the state name.

-10 meters/second^2 (roughly). The acceleration is integrated to get the velocity. which is, in turn, integrated to get the vertical position.

In figure 41, a *ZeroCrossingDetector* actor is used to detect when the vertical position of the ball is zero. This results in production of an event on the (discrete) output *bump*. Examining figure 38, you can see that this event triggers a state transition back to the same *free* state, but where the *initialVelocity* parameter is changed to reverse the sign and attenuate it by the *elasticity*. This results in the ball bouncing, and losing energy, as shown by the plot in figure 37.

As you can see from figure 38, when the position and velocity of the ball drop below a specified threshold, the state machine transitions to the state *stop*, which has no refinement. This results in the model producing no further output.

3.2 Numerical Precision and Zeno Conditions

The bouncing ball model of figures 37 and 38 illustrates an interesting property of hybrid system modeling. The *stop* state, it turns out, is essential. Without it, the time between bounces keeps decreasing, as does the magnitude of each bounce. At some point, these numbers get smaller than the representable precision, and large errors start to occur. If you remove the *stop* state from the FSM, and re-run the model, you get the result shown in figure 42. The ball, in effect, falls through the surface on which it is bouncing and then goes into a free-fall in the space below.

The error that occurs here illustrates some fundamental pitfalls with hybrid system modeling. The event detected by the *ZeroCrossingDetector* actor can be missed by the simulator. This actor works with the solver to attempt to identify the precise point in time when the event occurs. It ensures that the simulation includes a sample time at that time. However, when the numbers get small enough, numerical errors take over, and the event is missed.

A related phenomenon is called the Zeno phenomenon. In the case of the bouncing ball, the time between bounces gets smaller as the simulation progresses. Since the simulator is attempting to capture
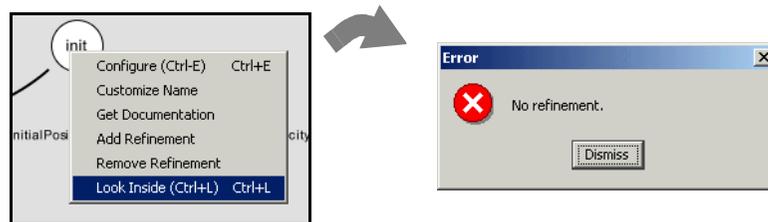
FIGURE 40. A state may or may not have a refinement, which specified the behavior of the model while the model i in that state. In this case, *init* has no refinement.
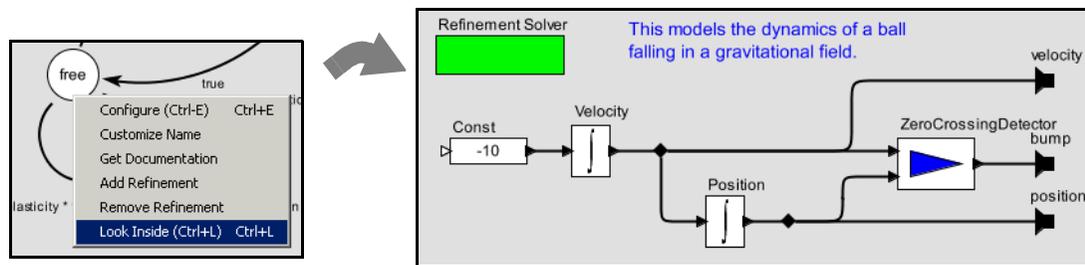
FIGURE 41. The refinement of the *free* state, shown here, is a continuous-model representing the laws of gravity.

every bounce event with a time step, we could encounter the problem where the number of time steps becomes infinite over a finite time interval. This makes it impossible for time to advance. In fact, in theory, the bouncing ball example exhibits this Zeno phenomenon. However, numerical precision errors take over, since the simulator cannot possibly keep decreasing the magnitude of the time increments.

The lesson is that some caution needs to be exercised when relying on the results of a simulation of a hybrid system. Use your judgement.

3.3 Constructing Modal Models

A modal model is a component in a larger continuous-time model. You can create a modal model by dragging one in from the *utilities* library. By default, it has no ports. To make it useful, you will need to add ports. The mechanism for doing that is identical to adding ports to a composite model, and is explained in section 2.4. Figure 37 shows a top-level continuous-time model with a single modal model that has been renamed *Ball Model*. Three output ports have been added to that modal model, but only the top one is used. It gives the vertical distance of the ball from the surface on which it bounces.

If you create a new modal model by dragging it in from the *utilities* library, create an output port and name it *output*, and then look inside, you will get an FSM editor like that shown in figure 43. Note that the output port is (regrettably) located at the upper left, and is only partially visible. The annotation text suggests delete once you no longer need it. You may want to move the port to a more reasonable location (where it is visible).

The output port that you created is in fact indicated in the state machine as being both an output and input port. The reason for this is that guards in the state machine can refer to output values that are produced on this port by refinements. In addition, the output actions of a transition can assign an output value to this port. Hence, the port is, in fact, both an output and input for the state machine.

To create a finite-state machine like that in figure 38, drag in states (white circles), or click on the state icon in the toolbar. You can rename these states by right clicking on them and selecting "Customize Name". Choose names that are pertinent to your application. In figure 38, there is an *init* state for initialization, a *free* state for when the ball is in the air, and a *stop* state for when the ball is no longer bouncing. You must specify the initial state of the FSM by right clicking on the background of the FSM Editor, selecting "Edit Parameters", and specifying an initial state name, as shown in figure 39. In that figure, the initial state is named *init*.
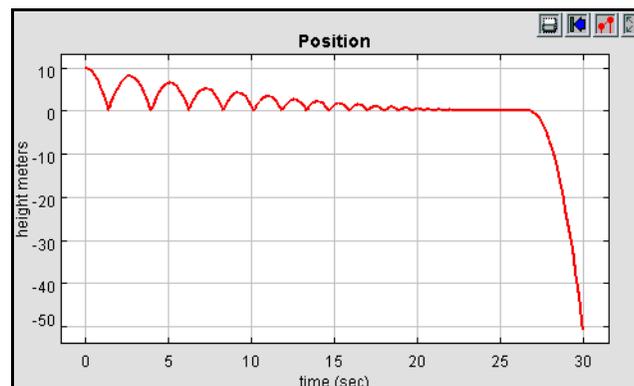
FIGURE 42. Result of running the bouncing ball model without the *stop* state.

3.3.1 Creating Transitions

To create transitions, you must hold the control button on the keyboard while clicking and dragging from one state to the next (a transition can also go back to the same state). The handles on the transition can be used to customize its curvature and orientation. Double clicking on the transition (or right clicking and selecting "Configure") allows you to configure the transition. The dialog for the transition from *init* to *free* is shown in Figure 44. In that dialog, we see the following:

- The guard expression is *true*, so this transition is always enabled. The transition will be taken as soon as the model begins executing. A guard expression can be any boolean-valued expression that depends on the inputs, parameters, or even the outputs of any refinement of the current state (see below). Thus, this transition is used to initialize the model.

- The output actions are empty, meaning that when this transition is taken, no output is specified. This parameter can have a list of assignments of values to output ports, separated by semicolons. Those values will be assigned to output ports when the transition is taken.
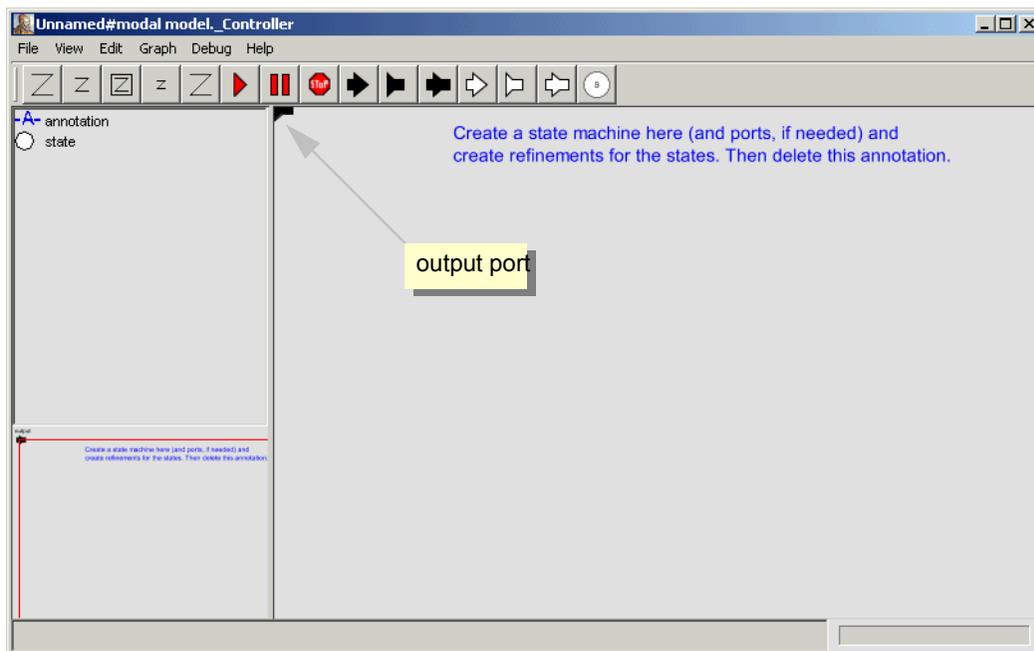
FIGURE 43. Inside of a new modal model that has had a single output port added.
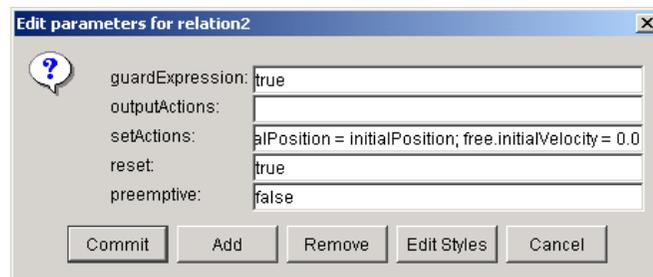
FIGURE 44. Transition dialog for the transition from *init* to *free* in Figure 37.

- The set actions field contains the following statements:

```
free.initialPosition = initialPosition; free.initialVelocity = 0.0
```

The "free" in these expressions refers to the mode refinement in the *free* state. Thus, *free.initialPosition* is a parameter of that mode refinement. Here, its value is assigned to the value of the parameter *initialPosition*. The parameter *free.initialVelocity* is set to zero.

- The *reset* parameter is set to *true*, meaning that the destination mode refinement will be initialized when the transition is taken.

- The *preemptive* parameter is set to *false*. In this case, it makes no difference, since the *init* state has no refinement. Normally, if a transition out of a state is enabled and *preemptive* is *true*, then the transition will be taken without first executing the refinement. Thus, the refinement will not affect the outputs of the modal model.

A state may have several outgoing transitions. However, it is up to the model builder to ensure that at no time does more than one guard on these transitions evaluate to true. In other words, HyVisual does not allow nondeterministic state machines, and will throw an exception if it encounters one.

3.3.2 Creating Refinements

Both states and transitions can have *refinements*. To create a refinement, right click on the state or transition, and select "Add Refinement." You will see a dialog like that in figure 46. As shown in the figure, you will be offered the alternatives of a "Continuous Time Refinement" or a "State Machine Refinement." The first of these provides a continuous-time model as the refinement. The second provides another finite state machine as the refinement. In the former case (the default), an almost blank refinement model will open, as shown in the figure. As before, the output port will appear in an inconvenient location. You will almost certainly want to move it to a more convenient location.

You can also create refinements for transitions, but these have somewhat different behavior. They will execute exactly once when the transition is taken. For this reason, they offer an entirely different "solver." In fact, transition refinements will execute according to dataflow semantics. They are typi-
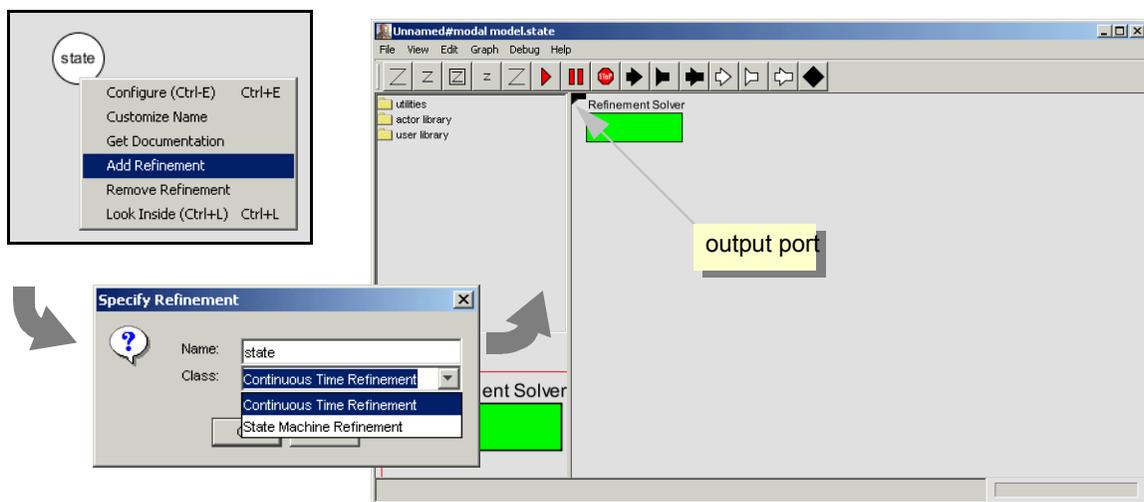
FIGURE 45. Adding a refinement to a state.

cally used to perform arithmetic computations that are too elaborate to be conveniently specified as an action on the transition.

Once you have created a refinement, you can look inside a state or transition. For the bouncing ball example, the refinement of the *free* state is shown in figure 41. This model exhibits certain key properties of refinements:

- Refinements must contain solvers, unlike composite actors. In this case, the solver is named "Refinement Solver" and is an instance of the CTEmbeddedDirector class.
- The refinement has the same ports as the modal model, and can read input value and specify output values. When the state machine is in the state of which this is the refinement, this model will be executed to read the inputs and produce the outputs.

3.4 Execution Semantics

The behavior of a refinement is simple. When the modal model is executed, the following sequence of events occurs:

- For any transitions out of the current state for which *preemptive* is *true*, the guard is evaluated. If exactly one such guard evaluates to *true*, then that transition is chosen. The *output actions* of the transition are executed, and the *refinements* of the transition (if any) are executed, followed by the *set actions*.
- If no preemptive transition evaluated to true, then the refinement of the current state, if there is one, is evaluated at the current time step.
- Once the refinement has been evaluated (and it has possibly updated its output values), the guard expressions on all the outgoing transitions of the current state are evaluated. If none is true, the execution is complete. If one is true, then that transition is taken. If more than one is true, then an exception is thrown (the state machine is nondeterministic). What it means for the transition to be "taken" is that its *output actions* are executed, its *refinements* (if any) are executed, and its *set actions* are executed.
- If *reset* is true on a transition that is taken, then the refinement of the destination mode (if there is one) is initialized.

There is a subtle distinction between the *output actions* and the *set actions*. The intent of these two fields on the transition is that *output actions* are used to define the values of output ports, while *set actions* are used to define state variables in the refinements of the destination modes. The reason that these two actions are separated is that while solving a continuous-time system of equations, the solver may speculatively execute models at certain time steps before it is sure what the next time step will be. The *output actions* make no permanent changes to the state of the system, and hence can be executed during this speculative phase. The *set actions*, however, make permanent changes to the state variables of the destination refinements, and hence are not executed during the speculative phase.

4. Using the Plotter

Several of the plots shown above have flaws that can be fixed using the features of the plotter. For instance, the plot shown in figure 16 has the default (uninformative) title, the axes are not labeled, and there is no legend.

The *TimedPlotter* actor has some pertinent parameters, shown in figure 46. The *fillOnWrapup* parameter specifies whether the plot should fill the available screen area when the model completes

execution. The default value is *true*, and for finite executions, this is often the most useful value. The *legend* parameter can be used to identify signals. It is a comma-separated listed of names that will be attached to signals. With the values shown in figure 46, the resulting plot looks like figure 47. The *startingDataset* parameter is used when two actors share the same plot, and is beyond the scope of this discussion.

The plot in figure 47 is better, but it is still missing useful information. To control more precisely the visual appearance of the plot, click on the second button from the right in the row of buttons at the top right of the plot. This button brings up a format control window. It is shown in figure 48, filled in

FIGURE 46. Parameters of the *TimedPlotter* actor.
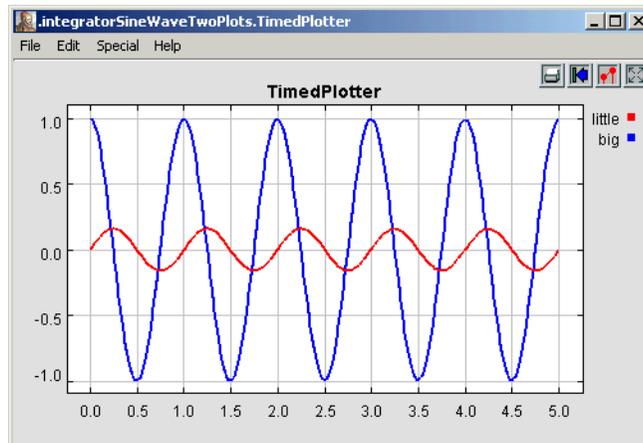
FIGURE 47. A plot with a legend.
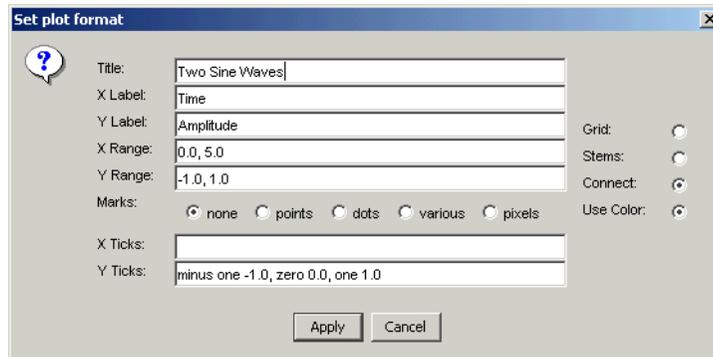
FIGURE 48. Format control window for a plot.

with values that result in the plot shown in figure 49. Most of these are self-explanatory, but the fol-lowing pointers may be useful:

- The grid is turned off to reduce clutter.
- Titles and axis labels have been added.
- The X range and Y range are determined by the fill button at the upper right of the plot.
- Stem plots can be had by clicking on "Stems."
- Individual sample points can be shown by clicking on "dots."
- Connecting lines can be eliminated by deselecting "connect."
- The Y axis label has been changed to text rather than numbers. This is done by entering the follow-ing in the Y Ticks field:

```
"minus one" -1.0, zero 0.0, one 1.0
```

The syntax in general is:

label value, label value, ...

where the label is any string (enclosed in quotation marks if it includes spaces), and the value is a num-ber.

5. Expressions

In HyVisual, models specify computations by composing actors. Many computations, however, are awkward to specify this way. A common situation is where we wish to evaluate a simple algebraic expression, such as "$\sin(2\pi(x-1))$." It is possible to express this computation by composing actors in a block diagram, but it is far more convenient to give it textually.

The expression language provides infrastructure for specifying algebraic expressions textually and for evaluating them. The expression language is used to specify the values of parameters, guards and actions in state machines, and for the calculation performed by the *Expression* actor. In fact, the expression language is part of the generic infrastructure in Ptolemy II, upon which HyVisual is built.
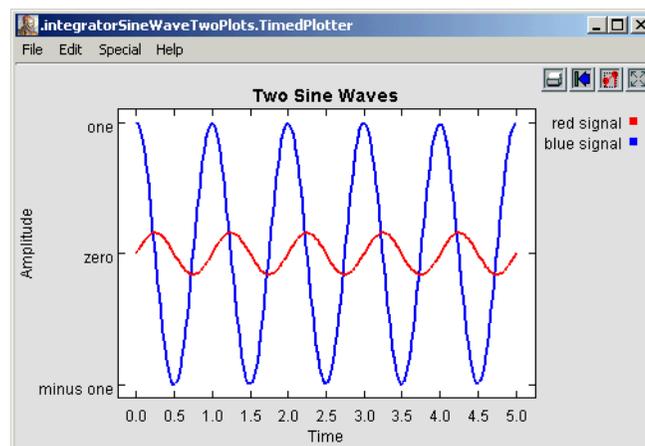
FIGURE 49. Still better labeled plot.

5.1 Simple Arithmetic Expressions

5.1.1 Constants and Literals

The simplest expression is a constant, which can be given either by the symbolic name of the constant, or by a literal. By default, the symbolic names of constants supported are PI, pi, E, e, true, false, i, and j. for example,

```
PI/2.0
```

is a valid expression that refers to the symbolic name "PI" and the literal "2.0." The constants i and j are complex numbers with value equal to 0.0 + 1.0i.

Numerical values without decimal points, such as "10" or "-3" are integers. Numerical values with decimal points, such as "10.0" or "3.14159" are doubles. Integers followed by the character "l" (el) or "L" are long integers. Integers beginning with a leading "0" are octal numbers. Integers beginning with a leading "0x" are hexadecimal numbers. For example, "012" and "0xA" are both the integer 10. In releases later than Ptolemy II 2.0.1, but not including 2.0.1 itself, integers followed by "ub" or "UB" are unsigned bytes, as in "5ub". Literal string constants are also supported. Anything between quotes, "...", is interpreted as a string constant.

A complex is defined by appending an "i" or a "j" to a double for the imaginary part. This gives a purely imaginary complex number which can then leverage the polymorphic operations in the Token classes to create a general complex number. Thus "2 + 3i" will result in the expected complex number. You can optionally write this "2 + 3*i".

5.1.2 Summary of Supported Types

The types currently supported in the expression language are *boolean*, *unsigned byte*, *complex*, *fixedpoint*, *double*, *int*, *long*, *array*, *matrix*, *record*, and *string*. The composite types, array, matrix, and record, are described below in section 5.3. Note that there is no float (as yet). Use double or int instead. A long is defined by appending an integer with "l" (lower case L) or "L", as in Java. A fixed point number is defined using the "fix" function, as will be explained below in section 5.5.

5.1.3 Variables

Expressions can contain references to variables within the *scope* of the expression. For example,

```
PI*x/2.0
```

is valid if "x" is a variable in scope. In the context of Ptolemy II models, the variables in scope include all parameters defined at the same level of the hierarchy or higher. So for example, if an actor has a parameter named "x" with value 1.0, then another parameter of the same actor can have an expression with value "PI*x/2.0", which will evaluate to $\pi/2$.

Consider a parameter P in actor X which is in turn contained by composite actor Y. The scope of an expression for P includes all the parameters contained by X and Y, plus those of the container of Y, its container, etc. That is, the scope includes any parameters defined above in the hierarchy.

You can add parameters to actors (composite or not) by right clicking on the actor, selecting "Configure" and then clicking on "Add", or by dragging in a parameter from the *utilities* library. Thus, you

can add variables to any scope, a capability that serves the same role as the "let" construct in many programming languages.

5.1.4 Operators

The arithmetic operators are +, −, *, /, ^, and %. Most of these operators operate on most data types, including matrices. The ^ operator computes "to the power of" where the power can only be an integer. The bitwise operators are &, |, #, and ~. They operate on integers, where & is bitwise and, ~ is bitwise not, and | is bitwise or, and # is bitwise exclusive or (after MATLAB).

The relational operators are <, <=, >, >=, == and !=. They return booleans. Boolean-valued expressions can be used to give conditional values. The syntax for this is

```
boolean ? value1 : value2
```

If the boolean is true, the value of the expression is `value1`; otherwise, it is `value2`.

The logical boolean operators are &&, ||, !, & and |. They operate on booleans and return booleans. The difference between logical && and logical & is that & evaluates all the operands regardless of whether their value is now irrelevant. Similarly for logical || and |. This approach is borrowed from Java.

The << and >> operators performs arithmetic left and right shifts respectively. The >>> operator performs a logical right shift, which does not preserve the sign.

5.1.5 Comments

In expressions, anything inside /*...*/ is ignored, so you can insert comments.

5.2 Uses of Expressions

5.2.1 Parameters

The values of most parameters of actors can be given as expressions[1]. The variables in the expression refer to other parameters that are in scope, which are those contained by the same container or some container above in the hierarchy. Adding parameters to actors is straightforward, as explained in section 2.2.4.

5.2.2 Port Parameters

It is possible to define a parameter that is also a port. Such a *PortParameter* provides a default value, which is specified like the value of any other parameter. When the corresponding port receives data, however, the default value is overridden with the value provided at the port. Thus, this object function like a parameter and a port. The current value of the PortParameter is accessed like that of any other parameter. Its current value will be either the default or the value most recently received on the port.

A PortParameter might be contained by an atomic actor or a composite actor. To put one in a composite actor, drag it into a model from the *utilities* library, as shown in figure 50. The resulting icon is actually a combination of two icons, one representing the port, and the other representing the parame-

---

1. The exceptions are parameters that are strictly string parameters, in which case the value of the parameter is the literal string, not the string interpreted as an expression, as for example the *function* parameter of the *TrigFunction* actor, which can take on only "sin," "cos," "tan", "asin", "acos", and "atan" as values.

ter. These can be moved separately, but doing so might create confusion, so we recommend selecting both by clicking and dragging over the pair and moving both together.

To be useful, a PortParameter has to be given a name (the default name, "portParameter," is not very compelling). To change the name, right click on the icon and select "Customize Name," as shown in figure 50. In the figure, the name is set to "noiseLevel." Then set the default value by either double clicking or selecting "Configure." In the figure, the default value is set to 10.0.

In a continuous-time model, if a PortParameter is supplied with discrete data at the port, then it must be declared DISCRETE. To do this, create a parameter in the PortParameter with name *signalType* and value "DISCRETE" (with the quotation marks).

5.2.3 Expression Actor

The *Expression* actor is a particularly useful actor found in the *math* library. By default, it has one output an no inputs, as shown in Figure 51(a). The first step in using it is to add ports, as shown in (b) and (c), resulting in a new icon as shown in (d). Note: In (c) when you click on Add, you will be prompted for a Name (pick one) and a Class. Leave the Class entry blank and click OK. You then specify an expression using the port names, as shown in (e), resulting in the icon shown in (f).

5.2.4 State Machines

Expressions give the guards for state transitions, as well as the values used in actions that produce outputs and actions that set values of parameters in the refinements of destination states. This mechanism was explained earlier.
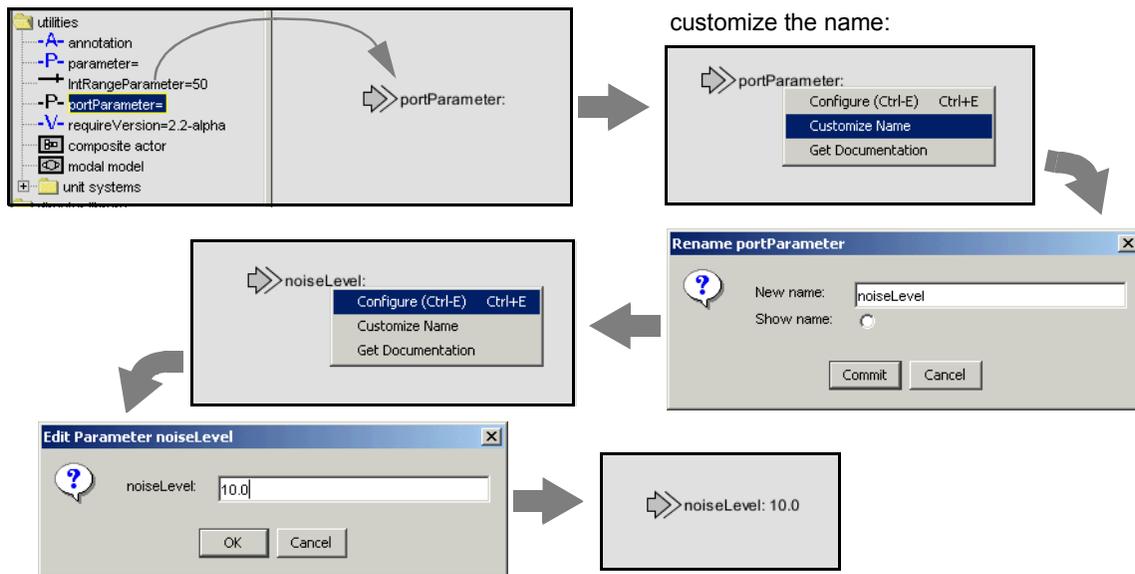
FIGURE 50. A *portParameter* is both a port and a parameter. To use it in a composite actor, drag it into the actor, change its name to something meaningful, and set its default value.

5.3 Composite Data Types

5.3.1 Arrays

Arrays are specified with curly brackets, e.g., "{1, 2, 3}" is an array of integers, while "{`"x"`, `"y"`, `"z"`}" is an array of strings. An array is an ordered list of tokens of any type, with the only constraint being that the elements all have the same type. Thus, for example, "{1, 2.3}" is illegal because the first element is an integer and the second is a double. The elements of the array can be given by expressions, as in the example "{2*pi, 3*pi}." Arrays can be nested; for example, "{{1, 2}, {3, 4, 5}}" is an array of arrays of integers.

5.3.2 Matrices

In HyVisual, arrays are ordered sets of tokens. Ptolemy II also supports matrices, which are more specialized than arrays. They contain only primitive types, currently *boolean*, *complex*, *double*, *fixed-point*, *int*, and *long*. Matrices cannot contain arbitrary tokens, so they cannot, for example, contain matrices. They are intended for data intensive computations.

Matrices are specified with square brackets, using commas to separate row elements and semicolons to separate rows. E.g., "[1, 2, 3; 4, 5, 5+1]" gives a two by three integer matrix (2 rows and 3 columns). Note that an array or matrix element can be given by an expression, but all elements must have the same type, and that type must be one of the types for which matrices are defined. A row vector can be given as "[1, 2, 3]" and a column vector as "[1; 2; 3]". Some MATLAB-style array constructors are
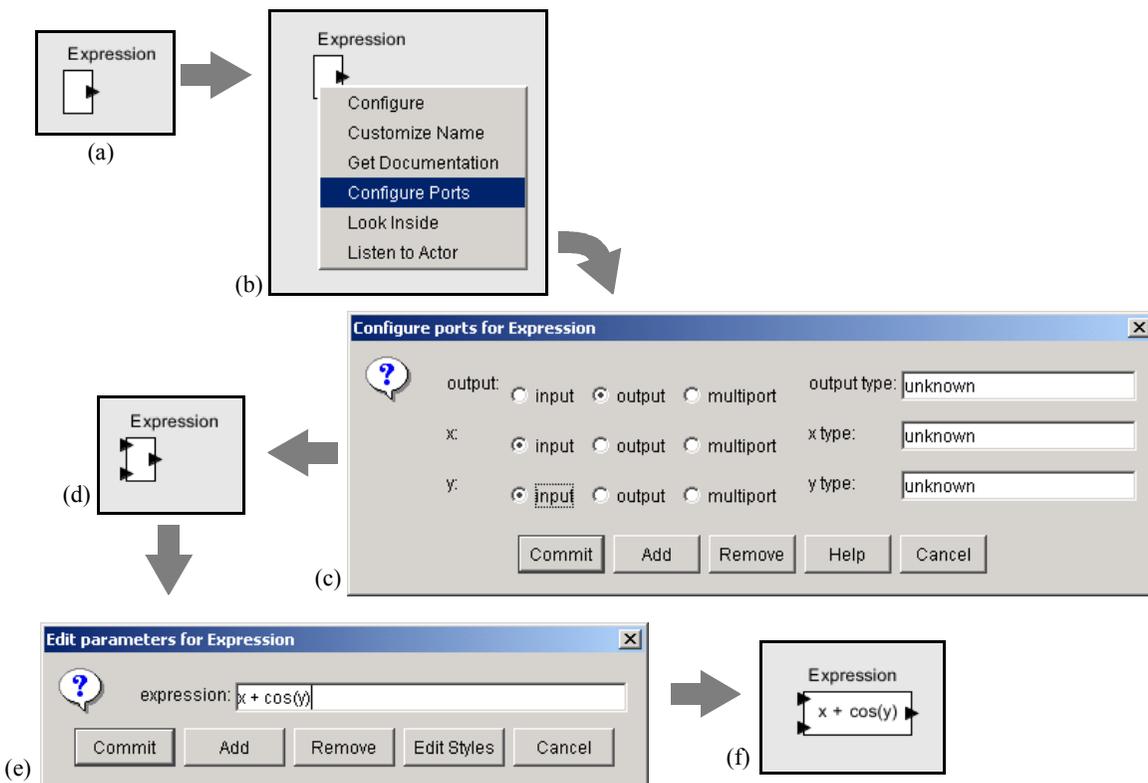
FIGURE 51. Illustration of the *Expression* actor.

supported. For example, "[1:2:9]" gives an array of odd numbers from 1 to 9, and is equivalent to "[1, 3, 5, 7, 9]." Similarly, "[1:2:9; 2:2:10]" is equivalent to "[1, 3, 5, 7, 9; 2, 4, 6, 8, 10]." In the syntax "[*p*:*q*:*r*]", *p* is the first element, *q* is the step between elements, and *r* is an upper bound on the last element. That is, the matrix will not contain an element larger than *r*.

Reference to matrices have the form "*name*(*n*, *m*)" where *name* is the name of a matrix variable in scope, *n* is the row index, and *m* is the column index. Index numbers start with zero, as in Java, not 1, as in MATLAB.

5.3.3 Records

A record token is a composite type where each element is named, and each element can have a distinct type. Records are delimited by curly braces, with each element given a name. For example, "{a=1, b="foo"}" is a record with two elements, named "a" and "b", with values 1 (an integer) and "foo" (a string), respectively. The value of a record element can be an arbitrary expression, and records can be nested (an element of a record token may be a record token).

Fields of records may be accessed using the period operator. For example,

```
{a=1,b=2}.a
```

yields 1. You can optionally write this as if it were a method call:

```
{a=1,b=2}.a()
```

5.4 Functions and Methods

5.4.1 Functions

The expression language includes an extensible set of functions, such as sin(), cos(), etc. The functions that are built in include all static methods of the java.lang.Math class and the ptolemy.data.expr.UtilityFunctions class. This can easily be extended by registering another class that includes static methods. The functions currently available are shown in Figures 52 and 53, with the argument types and return types[1].

One slightly subtle function is the random() function shown in Figure 52. It takes no arguments, and hence is written "random()". It returns a random number. However, this function is evaluated only when the expression within which it appears is evaluated. The result of the expression may be used repeatedly without re-evaluating the expression. The random() function is not called again. Thus, for example, if the *value* parameter of the *Const* actor is set to "random()", then its output will be a random constant, i.e., it will not change on each firing.

The property() function accesses system properties by name. Some possibly useful system properties are:

• ptolemy.ptII.dir: The directory in which Ptolemy II is installed.
• ptolemy.ptII.dirAsURL: The directory in which Ptolemy II is installed, but represented as a URL.
• user.dir: The current working directory, which is usually the directory in which the current executable was started.

---

1. At this time, in release 2.0, the types must match exactly for the expression evaluator to work. Thus, "sin(1)" fails, because the argument to the sin() function is required to be a double.

function	argument type(s)	return type	description
abs	double	double	absolute value
abs	int	int	absolute value
abs	long	long	absolute value
acos	double	double	arc cosine
asin	double	double	arc sine
atan	double	double	arc tangent
atan2	double, double	double	angle of a vector
ceil	double	double	ceiling function
cos	double	double	cosine
exp	double	double	exponential function (e^argument)
floor	double	double	floor function
IEEEremainder	double, double	double	remainder after division
log	double	double	natural logarithm
max	double, double	double	maximum
max	int, int	int	maximum
max	long, long	long	maximum
min	double, double	double	minimum
min	int, int	int	minimum
min	long, long	long	minimum
pow	double, double	double	first argument to the power of the second
random		double	random number between 0.0 and 1.0
rint	double	double	round to the nearest integer
round	double	long	round to the nearest integer
sin	double	double	sine function
sqrt	double	double	square root
tan	double	double	tangent function
toDegrees	double	double	convert radians to degrees
toRadians	double	double	convert degrees to radians

FIGURE 52. Functions available to the expression language from the java.lang.Math class.

5.4.2 Methods

Every element and subexpression in an expression represents an instance of the Token class in Ptolemy II (or more likely, a class derived from Token). The expression language supports invocation of any method of a given token, as long as the arguments of the method are of type Token and the return type is Token (or a class derived from Token, or something that the expression parser can easily convert to a token, such as a string, double, int, etc.). The syntax for this is (*token*).*methodName*(*args*), where *methodName* is the name of the method and *args* is a comma-separated set of arguments. Each argument can itself be an expression. Note that the parentheses around the *token* are not required, but might be useful for clarity. As an example, the ArrayToken class has a getElement(int) method, which can be used as follows:

```
{1, 2, 3}.getElement(1)
```

This returns the integer 2. Another useful function of array token is illustrated by the following example:

```
{1, 2, 3}.length()
```

function	argument type(s)	return type	description
constants	none	record	Record identifying all the globally defined constants in the expression language.
freeMemory	none	long	Return the approximate number of bytes available for future memory allocation.
gaussian	double, double	double	Gaussian random variable with the specified mean, and standard deviation
gaussian	double, double, int, int	double matrix	Gaussian random matrix with the specified mean, standard deviation, rows, and columns
property	string	string	Return a system property with the specified name from the environment, or an empty string if there is none.
readFile	string	string	Get the string text in the specified file. Return an empty string if the file is not found.
readMatrix	string	double matrix	Read a file that contains a matrix of reals in MATLAB notation.
repeat	int, general	array	Create an array by repeating the specified token the specified number of times.
totalMemory	long	none	Return the approximate number of bytes used by current objects plus those available for future object allocation.
findFile	string	string	Return an absolute file name given one that is relative to the user directory or the classpath.

FIGURE 53. Functions available to the expression language from the ptolemy.data.expr.UtilityFunctions class. This class is still at a preliminary stage, and the function it provides will grow over time.

which returns the integer 3.

The MatrixToken classes have three particularly useful methods, illustrated in the following examples:

```
[1, 2; 3, 4; 5, 6].getRowCount()
```

which returns 3, and

```
[1, 2; 3, 4; 5, 6].getColumnCount()
```

which returns 2, and

```
[1, 2; 3, 4; 5, 6].toArray()
```

which returns {1, 2, 3, 4, 5, 6}. The latter function can be particularly useful for creating arrays using MATLAB-style syntax. For example, to obtain an array with the integers from 1 to 100, you can enter:

```
[1:1:100].toArray()
```

The get() method of RecordToken accesses a record field, as in the following example:

```
{a=1, b=2}.get("a")
```

which returns 1.

The Token classes from the data package form the primitives of the language. For example the number 10 becomes an IntToken with the value 10 when evaluating an expression. Normally this is invisible to the user. The expression language is object-oriented, of course, so methods can be invoked on these primitives. A sophisticated user, therefore, can make use of the fact that "10" is in fact an object to invoke methods of that object.

In particular, the convert() method of the Token class might be useful, albeit a bit subtle in how it is used. For example:

```
double.convert(1)
```

creates a DoubleToken with value 1.0. The variable *double* is a built-in constant with type double. The convert() method of DoubleToken converts the argument to a DoubleToken, so the result of this expression is 1.0. A more peculiar way to write this is

```
(1.2).convert(1)
```

Any double constant will work in place of 1.2. Its value is irrelevant.

The convert() method supports only lossless type conversion. Lossy conversion has to be done explicitly via a function call.

5.5 Fixed Point Numbers

Ptolemy II includes a preliminary fixed point data type. We represent a fixed point value in the expression language using the following format:

```
fix(value, totalBits, integerBits)
```

Thus, a fixed point value of 5.375 that uses 8 bit precision of which 4 bits are used to represent the integer part can be represented as:

```
fix(5.375, 8, 4)
```

The value can also be a matrix of doubles. The values are rounded, yielding the nearest value representable with the specified precision. If the value to represent is out of range, then it is saturated, meaning that the maximum or minimum fixed point value is returned, depending on the sign of the specified value. For example,

```
fix(5.375, 8, 3)
```

will yield 3.968758, the maximum value possible with the (8/3) precision.

In addition to the fix() function, the expression language offers a quantize() function. The arguments are the same as those of the fix() function, but the return type is a DoubleToken or DoubleMatrixToken instead of a FixToken or FixMatrixToken. This function can therefore be used to quantize double-precision values without ever explicitly working with the fixed-point representation.

To make the FixToken accessible within the expression language, the following functions are available:

• To create a single FixPoint Token using the expression language:
```
fix(5.34, 10, 4)
```

This will create a FixToken. In this case, we try to fit the number 5.34 into a 10 bit representation with 4 bits used in the integer part. This may lead to quantization errors. By default the round quantizer is used.

• To create a Matrix with FixPoint values using the expression language:
```
fix([ -.040609, -.001628, .17853 ], 10,  2)
```
This will create a FixMatrixToken with 1 row and 3 columns, in which each element is a Fix-Point value with precision(10/2). The resulting FixMatrixToken will try to fit each element of the given double matrix into a 10 bit representation with 2 bits used for the integer part. By default the round quantizer is used.

• To create a single DoubleToken, which is the quantized version of the double value given, using the expression language:
```
quantize(5.34, 10, 4)
```

This will create a DoubleToken. The resulting DoubleToken contains the double value obtained by fitting the number 5.34 into a 10 bit representation with 4 bits used in the integer part. This may lead to quantization errors. By default the round quantizer is used.

• To create a Matrix with doubles quantized to a particular precision using the expression language:
```
quantize([ -.040609, -.001628, .17853 ], 10,  2)
```

This will create a DoubleMatrixToken with 1 row and 3 columns. The elements of the token are obtained by fitting the given matrix elements into a 10 bit representation with 2 bits used for the integer part. Instead of being a fixed point value, the values are converted back to their double representation and by default the round quantizer is used.